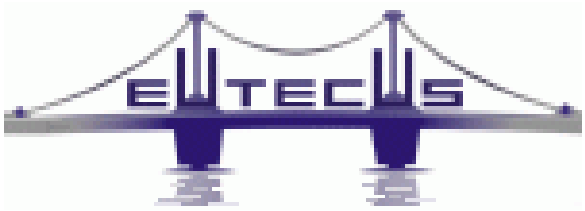

Bi-i Vision System

Programming Guide

Eutecus, Inc.

Bi-i Vision System: Programming Guide

3.1.36

Eutecus, Inc.

1936 University Ave., Suite 360

Berkeley, CA, 94704

Phone: (510) 540-9603

Fax: (510) 649-7808

http://www.eutecus.com

info@eutecus.com

Copyright © 2004-2007 Eutecus, Inc.

Table of Contents

1. Introduction 1
2. The AMC Language 2
    2.1. Introduction 2
    2.2. AMC Compiler 2
    2.3. Running AMC Programs 2
    2.4. General Rules 3
    2.5. Argument Types 3
    2.6. Address Spaces 5
    2.7. Data Structures 5
        2.7.1. Images and Other 2D Arrays 5
        2.7.2. Vectors 6
        2.7.3. Templates 6
    2.8. Basic AMC Code 6
    2.9. Image handling 6
    2.10. Flow Control 7
        2.10.1. Loop 7
        2.10.2. Calling Subroutines 7
        2.10.3. Tiling 8
        2.10.4. Definitions And Conditional Compilation 9
    2.11. Avi handling 10
    2.12. Frame Grabber 11
    2.13. Image Acquisition from the Ibis Sensor 12
    2.14. ACE16k 12
        2.14.1. Image And Template I/O 13
        2.14.2. Logical Operation 13
        2.14.3. Template Execution 14
        2.14.4. Image Acquisition 14
    2.15. Image Processing Library (IPL) 14
    2.16. Profiling 15
3. AMCedit 16
    3.1. Editing AMC codes 16
        3.1.1. The Editor Window 16
        3.1.2. The Info-tree Window 17
        3.1.3. Status Bar 18
    3.2. Running the AMC 18
        3.2.1. Selecting the Device 18
        3.2.2. Selecting the Workspace 19
        3.2.3. Compiling the AMC 19
        3.2.4. Running the ABC 19
    3.3. Menu Items, Toolbar Buttons and Shortcut Keys 19
4. The API 23
    4.1. Running an Application 23
    4.2. Receiving Messages 25
    4.3. Sending Messages 26
    4.4. Data Transfer 26
        4.4.1. Data Representation 26
        4.4.2. Data Transfer Messages 27
        4.4.3. Direct Transfer 28
    4.5. Cooperation with AMC and SDK 28
    4.6. .NET Components 29
        4.6.1. Installing the Components 29
        4.6.2. Usage of the Components 30
    4.7. Parameter generation (ParamTool utility) 32
        4.7.1. Special comments 33
        4.7.2. Parameter Handling Class 34

4.7.3. Example 36
5. The Bi-i SDK 41
5.1. Getting started 41
5.1.1. The Software Environment 41
5.1.2. Running an Application 41
5.1.3. Memory Usage 41
5.1.4. Interrupt Vectors 42
5.1.5. Build Options 42
5.1.6. Namespaces 43
5.1.7. Initialization and Closing 43
5.1.8. Logging 43
5.1.9. Stop and Pause Status 44
5.1.10. Checking the connection 44
5.2. General Purpose I/O interface (GPIO) 44
5.2.1. GPIO v301 component 45
5.3. Host Communication 45
5.3.1. Types of Communication 46
5.4. TIbis 48
5.4.1. Framerate 48
5.4.2. Acquiring Images the Simple Way 49
5.4.3. Parallel Operations 49
5.4.4. B Mode 49
5.4.5. Subsampling 50
5.4.6. Rolling Shutter 50
5.4.7. Subwindows 52
5.4.8. Auto Shutter 53
5.4.9. Using the Stereo HRES 54
5.4.10. Flash control 55
5.4.11. Triggered shutter 55
5.5. TACE and TACE_IPL 56
5.5.1. I/O Interface 56
5.5.2. Internal Image Transfer 60
5.5.3. Image Initialization 60
5.5.4. Operations on Images 60
5.5.5. Capturing the Optical Input 63
5.5.6. Image Processing Library (IPL) 63
5.6. L2 Cache 64
5.6.1. L2 Mode Selection 65
5.6.2. Memory Coherence 65
5.7. Video Analytics FrontEnd 69
5.8. Laser control 70
5.9. Debug Bi-i application with JTAG 71

List of Figures

3.1. AMC Syntax dialog 16
3.2. The Select Device dialog 18
4.1. Add the components 31
4.2. Link the Components 32
5.1. Memory Architecture of Bi-i System 64
5.2. Cache Coherence Problem 66
5.3. JTAG connectors in the Bi-i v2 72
5.4. JTAG connectors in the Bi-i v301 73

List of Tables

2.1. Free memory for AMC programs 5
3.1. Operations with the mouse on the info-tree 17
4.1. InfoParam to be passed depending on InfoType 24
4.2. The members of the union in the Bii_HostData structure 27
4.3. Messages received in the host application 28
4.4. Messages sent from the host application 28
5.1. Memory ranges on the Bi-i systems 42
5.2. Recommended libraries 43
5.3. Supported services on the Bi-i system 46
5.4. Horizontal patterns for Ibis5 subsampling 50

List of Examples

2.1. AMC compiling in a command prompt 2
2.2. AMC data types 4
2.3. Hello World application in AMC 6
2.4. Loading an image from the host computer in AMC 7
2.5. Initializing an image in AMC 7
2.6. Loop constructs in AMC 7
2.7. Calling a subroutine in AMC 8
2.8. Tiling in AMC 8
2.9. Conditional compilation in AMC 10
2.10. Reading an existing AVI file in AMC 10
2.11. Initializing an AVI file and writing frames to it in AMC 11
2.12. Getting images from the Hexium frame grabber in AMC 11
2.13. Getting images from a DirectShow source in AMC 12
2.14. Image acquisition from the Ibis sensor in AMC 12
2.15. Image I/O for ACE16k in AMC 13
2.16. Template I/O for ACE16k in AMC 13
2.17. Bit-wise logical operations on ACE16k in AMC 13
2.18. Continuous template execution in AMC 14
2.19. Iterative template execution with freezing mask in AMC 14
2.20. Image acquisition with ACE16k in AMC 14
2.21. Typical morphological procedure with IPL in AMC 15
2.22. Typical gray scale procedure with IPL in AMC 15
2.23. Measuring time in AMC 15
4.1. Calling Bii_Run 25
4.2. Receiving messages in a thread 26
4.3. Sample Parameter Comment block 33
4.4. Header Comment Sample, Enumeration Types 36
4.5. Header Comment Sample, Different range types 37
4.6. Header Comment Sample, Explicit data type parameters 38
4.7. Header Comment Sample, Matrix parameters 39
4.8. Header Comment Sample, Interface and Indirect parameters 40
5.1. The empty Bi-i SDK application 43
5.2. Bi-i SDK application with logging 44
5.3. Checking the stop status 44
5.4. Usage of GPIO interface 45
5.5. Usage of GPIO v301 with GPIO interface 45
5.6. Host Targets 46
5.7. Synchronous host communication 47
5.8. Asynchronous host communication 48
5.9. Taking an Image using operator >> on a TIbis object 49
5.10. Parallel operations on the Ibis5 sensor 49
5.11. B mode shutter on the Ibis5 sensor 50
5.12. Setting subsampling both directions on Ibis5 50
5.13. Rolling shutter mode on Ibis5 - USELESS CODE 51
5.14. Rolling shutter mode on Ibis5 52
5.15. Subwindows readout on Ibis5 53
5.16. Auto Shutter 54
5.17. Acquiring Images from the Stereo HRES 55
5.18. Setting flash 55
5.19. Triggered shutter 56
5.20. Setting the current IDs on ACE16k 58
5.21. Image transfer between the DSP and ACE16k with TACE 58
5.22. Template transfer with TACE 59
5.23. Loading templates from the host computer 59
5.24. Internal image copying with TACE 60

5.25. Image conversion with TACE 60
5.26. Initialization of image memories with TACE 60
5.27. Bit-wise logic operations with TACE 61
5.28. Continuous time template execution with TACEtransient 62
5.29. Iterative template execution with TACEtransient 62
5.30. Diffusion with TACE 62
5.31. Image capturing on ACE16k with TACE 63
5.32. Typical morphological operation with TACE_IPL 63
5.33. Typical gray scale operation with TACE_IPL 64
5.34. L2 Mode Selection 65
5.35. L2 Cache and Asynchronous Host Communication 68
5.36. L2 Cache and Parallel Reading from the Ibis sensor 68
5.37. L2 Cache and Direct Data Transfer 69
5.38. Using the TVAFrontEndF class 70
5.39. Using the TLaserController class 71

1. Introduction

This manual is a description of the different ways of Bi-i programming.

Chapter 2, *The AMC Language* describes the AMC language, which is the conventional way of Bi-i programming suitable for small applications. Chapter 3, *AMCedit* is an introduction to the AMCedit application, which is a syntax highlighting development environment for AMC. Choose AMC if you want to run relatively small image processing algorithms on Bi-i, such as testing the different interesting features of the ACE16K array processor, for example. No special programming skills are required to learn AMC and if you have some experience in computer programming with scripting languages (for example in Javascript or Matlab) the language will be familiar. All software tools required for AMC programming are installed with the Standard or the Development version of the Bi-i software.

Chapter 4, *The API* is a description of the Bi-i API, which is a programming interface to be used on the host computer for interfacing applications to Bi-i. Use the Bi-i API if the features of IVRun do not meet your requirements. For example, if a data acquisition program is needed that receives data from the Bi-i and stores it in a database. You can use the API with virtually any software development environment capable of handling dynamic libraries. The Bi-i API is installed with the Development version of the Bi-i software only.

Chapter 5, *The Bi-i SDK* is an overview of the Bi-i SDK, which is a collection of software libraries to be used when programming the Bi-i in C++. Use the SDK if you need a large, complex, thoroughly optimized Bi-i application, which is beyond the scope of AMC programming. You will need the Code Composer Studio application suite from Texas Instruments. C++ programming skills are required. The Bi-i SDK is installed with the Development version of the Bi-i software only.

In all cases it is assumed that you are familiar with the Bi-i hardware, Bi-i configuration and running applications on Bi-i. Consult the Bi-i Manual for more details.

2. The AMC Language

2.1. Introduction

AMC (Analogic Macro Code) is a simple programming language for Bi-i systems. All functionality of the Bi-i system can be accessed from an AMC program and can serve as a gentle introduction to programing on a Bi-i system.

2.2. AMC Compiler

The AMC source code can be compiled to Analogic Binary Code (ABC) with the AMC compiler (AMCComp.exe). The ABC code is an executable code on the Bi-i system. The AMC compiler is a stand-alone executable which has no graphical user interface. Use the AMC editor (AMCEdit.exe) to edit and compile the AMC code. AMC editor calls the AMC compiler when the appropriate menu item or toolbar button is selected.

The AMC compiler can be called from a command line as well with the following convention: a mandatory key must be added before the parameters separated with space character.

-e Engine file name with path

-d Device number

-i Input file name with path

-o Output file name with path

-l Log file name with path. If and error occured, the compiler writes log messages to this file

In all cases, if the path is not an absolute path, then it is treated as being relative to the path of the AMC compiler. The input and the engine or device parameters must be specified. If the log or output file name is not specified, these file names will be same as the input file name, and path of these are same the path of the input file.

Example 2.1. AMC compiling in a command prompt

```
AMCComp.exe –e engbiiv2_16k.bin –i test.amc –o test.abc –l test.log
or
amccomp.exe –d 1 –i test.amc –o test.abc –l test.log
```

Both of these samples compile the test.amc to test.abc for Bi-i v2 specified by device number 1 by the installation.

2.3. Running AMC Programs

The ABC code can be run by a host application, which downloads the ABC interpreter (specified by DSP program in the BiiConfig) and ABC code to the Bi-i system and starts it. The default execution environment is the IVRun_x86.exe (See in the InstantVision documentation).

It is very important that the following be set in the Properties dialog of IVConfig:

- Engine file in the 'Engine' dialog option.
- The DSP program is an AMC interpreter (ABCEx_v2_16k.out for Bi-i v2 and v301).
- 'ABC file required' check box is checked.

The details about BiiConfig can be found in the Bi-i Manual / Installation chapter / Bi-i Configuration section.

2.4. General Rules

An AMC program contains:

- AMC instructions (InstructionName Argument Argument ...)
- Labels (LabelName:)
- Conditions for compiling and defines (#ifdef..., #else, #endif, #define..., etc.).

An AMC program consists of lines. Semicolon (;) indicates comment, the comment is finished at the end of the line. Without comments a non-empty line may contain label and/or an instruction with parameters. General separator characters are space and tab collectively referred to as whitespace. The label, the instruction and the parameters must be separated from each other by whitespace, although a string parameter between double quotation marks may contain whitespace characters. The last character of a label must be a colon (:).

There is a special optional label: cleanup, which signals the cleanup code. If this label is specified and an error or stop message occurs, this code will be executed before the DSP program stops. This is important, for example, when a frame grabber is opened and sends images to DSP in an infinite cycle and the DSP receives a stop message or an error occurs. In this case the frame grabber can be stopped in this cleanup code.

The instructions have a fixed number of parameters except that one or more of the last parameters can have default values which is automatically appended by the compiler. AMC code is not case sensitive.

2.5. Argument Types

This section summarizes the basic data types supported in AMC. There are two groups of these types:

- Images and templates on ACE16k.
- Arguments on DSP.

Images and templates on ACE16k

C_TEM	Template memory.
C_LLM	Binary image memory.
C_LAM	Analog gray scale image memory.

Argument types on DSP

- Scalar types. The size of all scalar types are 4-byte in the memory.

M_GLM	Logical type scalar (0 - 1).
M_GAMB	Byte type scalar. Unsigned integer (0 - 255).
M_GAM	4-byte signed integer scalar.
M_GAMF	4-byte floating-point scalar.

- Vector types.

M_GLM_1D	Logical type 1D vector. The range of its elements is 0 - 1.
M_GAMB_1D	Byte type 1D vector. The range of its elements is 0 - 255 (unsigned integer).
M_GAM_1D	4-byte signed integer 1D vector.

M_GAMF_1D	4-byte floating-point 1D vector.

- 2D array types (Matrixes).

M_GLM_2D (M_LLM)	A two dimensional array of logical values (0 or 1). This is identical to M_LLM. M_LLM means the binary image.
M_GAMB_2D (M_LAM)	A two dimensional array of unsigned integers (0-255). This is identical to M_LAM. M_LAM means the gray scale image.
M_GAM_2D	A two dimensional array of 4-byte signed integers.
M_GAMF_2D	A two dimensional array of 4-byte floating-point numbers.
M_RGB	Color image. Actually, these are three M_LAM array with one header.

- M_TEM: Template memory (floating-point scalar array with 32 elements).
- Other types.

M_ADR	Memory address without type.
M_STR	Address of a string.
STR	String
LABEL	Name of an existing label in the AMC code.
NIL	Empty data, indicates that the operand is not given.

The details about these structures can be found in Section 2.7, “Data Structures”.

C_TEM, C_LLM and C_LAM types are referred as TEMx, LLMx and LAMx respectively where x is a number, thus TEM1 - TEM32 are template memories on ACE16K, LAM1 - LAM8 are gray scale image memories on ACE16K and LLM1 - LLM2 are binary image memories on ACE16K. There are some predefined symbols for all argument types on DSP memory. The size of predefined image locations is 128x128. M_GLM, M_GAMB, M_GAM and M_GAMF types can be referred to as direct data (input arguments only) or memory address. As direct data M_GLM, M_GAMB and M_GAM can be given in decimal or hexadecimal format with or without a +/- sign. M_GAMF can be given in floating point or exponential format. STR can be used as direct data only with or without double quotation mark. NIL data is referred to by the reserved word 'NIL'.

Example 2.2. AMC data types

```
TEM: TEM7
LLM: LLM1
LAM: LAM3
Integers: 23, -467
Floating point numbers: -24.529, 2.56e3
Strings: normal_text, "Text with white spaces in lower or UPPER CASE."
NIL:NIL
```

When referring to an address, an address specifier and an optional address type are used before the address itself. In case of a direct address the given address contains the data and the address specifier is $. In case of an indirect address the given address contains the location of the data and the address specifier is *$. The address type shows the type of the data at the given address. Valid address types are M_TEM, M_LLM, M_LAM, M_RGB, M_GLM, M_GAMB, M_GAM, M_GAMF, M_GLM_1D, M_GAMB_1D, M_GAM_1D, M_GAMF_1D, M_GLM_2D, M_GAMB_2D, M_GAM_2D, M_GAMF_2D, M_STR. The address can be given in decimal or hexadecimal format. The special address STCK refers to the current position of the stack pointer.

Note

This stack is not the stack for DSP program (function calling, etc.), this is declared for AMC flow control instructions (call, ret, etc.).

2.6. Address Spaces

In AMC addresses refer to physical addresses in the DSP memory. The AMC interpreter and the ABC code occupy different memory addresses depending on the Bi-i version. Only the Bi-i versions are supported. The remaining memory can be used freely by AMC programs (images, arrays vectors, etc.).

	Bi-i v2 and v301
Internal memory	0x000F0000 - 0x000FFFFF (64KB)
External memory	0x80000080 - 0x80FFFFFF (16MB - 0x80Byte) or 0x80000080 - 0x83FFFFFF (64MB - 0x80Byte)

Table 2.1. Free memory for AMC programs

The predefined arguments (LAMs, LLMs, etc.) are defined from the beginning of the free internal memory, therefore if a predefined address is used, then these memory addresses are overwritten.

2.7. Data Structures

2.7.1. Images and Other 2D Arrays

All images and 2D arrays start with a 5 word (20 byte) header that describes the type and size of the data. These values are represented by unsigned int values in the memory. The header structure is as follows:

1. Array type
2. Number of elements in a row
3. Number of rows
4. Size of a row in words
5. Size of the array without the header in bytes

In AMC, three types of images are used:

- The binary image type (M_LLM or M_GLM_2D). Each pixel occupies 1bit of memory. The value of the image type in the image header is 0.
- The grayscale image type (M_LAM or M_GAMB_2D) with 8 bit color depth, each pixel occupying 1 byte. The value of the image type in the image header is 1.
- The color image type (M_RGB). This data structure contains 1 image header and 3 gray scale image (M_LAM) fields for the 3 color channels red, green, blue respectively. The value of the image type in the image header is 2. The value of the image size field in the header means the size of a single color channel .

These structures are word aligned so the remainder space of the last word in a row is not used. The first pixel of the next row starts at the beginning of the next word. The other array types have same header and the element of these stored in 4 bytes. Below is an example for calculating the size of a 128x128 grayscale image in bytes.

```
Image size in bytes = 5*4 (header) +
   128 (height) * ((128 (width) + 3) / 4) (row size in words) * 4 =
   20 + 128 * ((128 + 3) / 4) * 4 = 16404 bytes
```

2.7.2. Vectors

Vectors are actually 1D array forms. The structure of header is different from header of 2D arrays, since the size of it is 3 words (12 bytes):

1. Vector type
2. Number of elements in the vector
3. Size of the vector without the header

For example, the size of a vector in bytes which contains 20 boolean elements is:

```
Vector size in bytes = 3*4byte (header) +
   ((20 (number of elements) + 31) / 32) (size in words) * 4 = 16 bytes
```

2.7.3. Templates

The template is a simple array, which contains 32 float values (128 bytes). The feedback matrix is denoted by the first nine values, the control matrix is denoted by the second nine values, the 19th value is the current, the 20th value is the BIAS weight, the 21th value is the template mode, and the last 11 values are the reference values in the ACE16K chip in the following order: Optical references (VIN, VPCH, BIAS_W), Weight references (VW-MAX, VW0, VWMIN, VGP) and Signal references (VXMAX, VX0, VXMIN, VDS).

2.8. Basic AMC Code

Each AMC program should contain and *end-of-program* marker (**end** instruction) otherwise the compiler will generate and error. Namely, the simplest AMC code is only an end instruction. In the simple 'Hello World' example below, you can see that the host application starts and immediately stops without any action.

Example 2.3. Hello World application in AMC

```
;Prints text to a text window in the host application.
host.print "Hello World!!!" 1

;End of AMC program. Stops the application.
end
```

When this AMC program is run, the string "Hello World!!!" appears in a window having ID=1 in the host application.

2.9. Image handling

The Bi-i software platform only supports the handling of BMP format images, so only these types of images are supported from AMC as well. There are several methods to deliver an image to the DSP's memory:

- Load the image from the host computer (and the image in DSP memory can be saved to the host computer)
- Get an image from the host application (and it can be displayed in the host application)
- Acquire an image with them Ibis CMOS sensor
- Acquire an image with ACE16K
- Read frames from an AVI file
- Acquire an image from a frame grabber device in the host computer
- Build an image in DSP memory

Example 2.4. Loading an image from the host computer in AMC

```
;Loads color image from the host computer to the DSP memory.
host.load.pic rgb.bmp $0x80010000 rgb

;Displays the image to the host application.
host.display $0x80010000 1
```

This code loads the rgb.bmp image in color format to the DSP memory, and displays it in the host application in window number 1.

Example 2.5. Initializing an image in AMC

```
;Initializing a 128*128 resolution gray scale image.
conv.gam_b.lam 120 LAM9 128 128

;Display it in the host application.
host.display LAM9 1
```

After running this program the LAM9 memory location contains a 128x128 resolution gray scale image. All pixel values of this image are 120 so we can see on the user interface of host application a homogeneous middle gray image.

2.10. Flow Control

In AMC we can create loops and subroutines to facilitate the processing of the acquired image sequence, counting, image processing pixel-by-pixel, branching, etc.

2.10.1. Loop

The loop construct in AMC is a special use of the jump instructions (jump, jumpc, jumpnc). The begining of a loop is a label. The loop variable is changed inside of loop core, and it should be initialized before loop is started. The condition checking can be anywhere inside the loop. For example we want to increase a counter variable and print this. The loop variable same as the counter variable.

Example 2.6. Loop constructs in AMC

```
mov.gam.gam 1 GAM1                  ;Initializing of loop variable.
BeginOfLoop:                        ;Begining of loop.
  host.print "Counter: %d" 1 GAM1   ;Print the counter.
sc.rel.equ 20 GAM1 GLM1             ;Check the loop condition.
sc.inc.gam GAM1                     ;Increment the loop variable.
jumpnc GLM1 BeginOfLoop             ;Conditional jumping.
```

The loop variable is the same as the counter variable (GAM1) and it is increased from 1 to 20. In the host application prints the following strings:

```
"Counter: 1"
"Counter: 2"
"Counter: 3"
...
...
"Counter: 19"
"Counter: 20"
```

2.10.2. Calling Subroutines

A subroutine is a part of program which can be called anywhere in the AMC program. The entry point of a subroutine is a label which identifies the subroutine for the call instruction. When the subroutine is called, the instruction pointer is put to the top of the stack and is set to the begining of the subroutine (to the specified label). Usually,

the end of a subroutine is a return instruction (ret, retc, retnc). When the return has occurred, the top of the stack is popped and the instruction pointer is set to this so the next instruction will be the one following the call instruction.

Example 2.7. Calling a subroutine in AMC

```
;Calling of 'func' subroutine.
call func

host.print "Func is returned." 2

;End of program.
end

;Subroutine
func:
   host.print "Func is called" 1
ret
```

The subroutine is identified by 'func' label which prints the text "Func is called" to host application.

Note

Normally subroutines are placed after 'end' instruction

2.10.3. Tiling

Tiling is a special loop construct in AMC for segmenting a large image into uniform smaller subimages (ACE16K size) with specified overlapping and merging these subimages into another large image after processing. This is very useful for processing a large image on the ACE16K. It is possible to process an arbitrary number of large images at the same time, hence multi-input multi-output algorithms can be defined in this way. The only requirement is that all of the images must be processed at the same time (in the same loop) and they all must have the same size.

The loop starts with a 'tile.loop.start' instruction. The 'tile.loop.start' instruction calculates the relative cutting position and the overlap. The 'tile.cutout' instruction cuts out a ACE16K sized image part specified by the latest 'tile.loop.start' instruction. The cut out part is saved into a new memory location. In case of a multi-input algorithm, there can be more 'tile.cutout' instructions. These instructions cut out chip size images from the same image location from different large images. Among the large images there can be both gray scale and binary images. Then, the chip sized image part(s) are processed. Finally, the processed chip sized image parts are merged together again with the 'tile.merge' instructions. The loop is closed with the 'tile.loop.end' instruction.

When the 'tile.loop.start' instruction is first executed, it assigns the upper left corner of the large image to be cut out. Then, the segmentation goes on from left to right from top to bottom.

Example 2.8. Tiling in AMC

```
;Loads the large images as gray scale type.
host.load.pic "large1.bmp" $0x80010000
host.load.pic "large2.bmp" $0x80030000

tile.loop.start 1 1   ;Minimal overlapping is 1, 1.
   tile.cutout $0x80010000 LAM9
   tile.cutout $0x80030000 LAM10

   ;Loop core.
   ;This is the place for processing of sub images

   tile.merge LAM9 $0x80050000
   tile.merge LAM10 $0x80070000
tile.loop.end
```

This particular code segments the input large images (location: $0x80010000, $0x80030000) and after processing merges the resulted sub images into the output large images (location: $0x80050000, $0x80070000).

2.10.4. Definitions And Conditional Compilation

It is possible to create and release compile-time constants or symbols with the #define and #undefine keywords. If a symbol is created, this symbol can be used as a predefined symbol. There are two possible ways to create a compile-time constant.

First, we want to define a constant symbol with a value, for example:

```
#define FloatValue 12.55
```

In the second case, we want to define a symbol for the conditional compilation. The value of the defined symbol will be similar to the symbol name. For example:

```
#define Condition
```

This is equivalent to

```
#define Condition Condition
```

If a symbol is not needed then we can release it. For example:

```
#undefine Condition
```

These definitions and the predefined symbols can be used for conditional compilation. Any part of a program can be included or omitted during the compilation depending on these symbols as conditions. The following keywords can be used for conditional compilation:

- #ifdef ’Symbol’: If the Symbol is defined the next part of AMC code will be included in the compilation.
- #ifndef ’Symbol’: If the Symbol is defined the next part of AMC code will be omitted from the compilation.
- #else: If the previous condition was true then the next part of AMC code will be omitted from compilation, and vice versa.
- #endif: All conditional compilation directives, such as #ifdef and #ifndef, must be matched with closing #endif directives prior to the end of file, otherwise, an error message is generated.

There are two predefined symbols the name for the Bi-i version

- 'Bi-iv2' :Defined if the device is a Bi-i v2.
- 'Bi-iv301:Defined if the device is a Bi-i v301.

  These symbols are equally useful for Bi-i v2 and Bi-i v301, because there is only one engine for both Bi-i version.

The next program illustrates how conditions can be used at compile-time:

Example 2.9. Conditional compilation in AMC

```
#define Symbol1
#ifdef Symbol1
[AMCCode1…]
#ifndef Symbol1
[AMCCode2…]
#else
[AMCCode3…]
#endif
[AMCCode4…]
#endif
[AMCCode5…]
```

This is equivalent to:

```
[AMCCode1…]
[AMCCode3…]
[AMCCode4…]
[AMCCode5…]
```

2.11. Avi handling

AMC programs can open AVI file for reading or writing or create new ones. To put frames into a previously opened or initialized AVI file, the properties of frame (width, height, bitcount) must be same as the properties of AVI file. A frame can only be written to the end of AVI file. If an AVI file is initialized a new empty AVI file is created on the file system of host computer with the specified path and name.

Example 2.10. Reading an existing AVI file in AMC

```
;Opens the specified AVI file for reading.
host.avi.open sample.avi AVI_READ GAM1
;Gets avi file length as loop condition.
host.avi.length GAM1 GAM2

mov.gam.gam 1 GAM3
loop:

   ;get GAM3-th frame from AVI file, and put it into DSP memory
   host.avi.getframe GAM1 GAM3 $0x80010000
   ;display image
   host.display $0x80010000 1

sc.rel.gam GAM3 GAM2 GLM1
sc.inc.gam GAM3
jumpnc GLM1 loop

cleanup:
   ;Closes AVI file.
   host.avi.close GAM1
```

Example 2.11. Initializing an AVI file and writing frames to it in AMC

In this case the AVI file can only be written. There is an image in the 0x80010000 address on the DSP memory (100*100, 24bit color depth).

```
;Initialzes a 100x100 24bit color depth AVI file
;with 25 frame/sec without compression..
host.avi.init sample.avi GAM1 100 100 24 FULL_FRAMES 25 100

loop:
   host.avi.putframe GAM1 $0x80010000
jump loop

cleanup:
host.avi.close GAM1
```

If reading an existing AVI file the initialization step should be replaced by AVI file opening in the previous example:

```
;Opens an existing AVI file.
host.avi.open sample.avi AVI_WRITE GAM1
```

Note

The closing of the AVI should be in the cleanup code, because the cleanup code will run even if the program is aborted due to an error.

2.12. Frame Grabber

Frame grabber devices on host computer can be used to grab image sequences. Only the Hexium frame grabber card and Microsoft Directshow devices are supported.

Example 2.12. Getting images from the Hexium frame grabber in AMC

```
;Opens Hexium frame grabber.
host.frgb.start GAM1 "HEXIUM" 0 0
;Sets signal type to PAL
host.frgb.signal GAM1 1 PAL

loop:
    ;Image resolution = 384*288*24bit from channel 0
    host.frgb.mov.pic GAM1 0 hexmode_384_288_24 $0x80010000 rgb
    ;Displays grabbed image
    host.display $0x80010000 1
jump loop

cleanup:
host.frgb.stop GAM1
```

The program grabs color images from the Hexium frame grabber and display those until the program is stopped.

Example 2.13. Getting images from a DirectShow source in AMC

```
;Opens a directshow device.
host.frgb.start GAM1 "DIRECTSHOW" 0 -1

loop:
     ;Grabs image from default channel and mode
     host.frgb.mov.pic GAM1 -1 0 $0x80010000
     host.display $0x80010000 1
jump loop

cleanup:
host.frgb.stop GAM1
```

The 'DirectShowTest.exe' utility shows all supported devices, channels and modes of connected DirectShow devices and those parameters can be written to the mode and channel arguments of 'host.frgb.mov.pic' instruction as well as the device argument of 'host.frgb.start' instruction.

Note

The closing of the framegrabber should be in the cleanup code, because the cleanup code will run even if the program is aborted due to an error.

2.13. Image Acquisition from the Ibis Sensor

Every Bi-i version contains one or two high resolution CMOS sensor(s). We can acquire images from these to the DSP memory with 'ibis.xxx' instructions.

Example 2.14. Image acquisition from the Ibis sensor in AMC

```
;Initializing of Ibis sensor.
ibis.init
;set image width in pixel
ibis.set ibis_nrof_pixels 800
;set image height in pixel
ibis.set ibis_nrof_lines 600
;set integration time in usec
ibis.set ibis_int_time 2500

loop:
    ;start image acquisition on Ibis sensor
    ibis.start
    ;wait for image acquisition ready
    ibis.wait
    ;read image from ibis
    ibis.read $0x80010000
    ;display image to 1.window
    host.display $0x80010000 1
jump loop
```

This particular program acquires an image sequence from the Ibis sensor. The program runs until the host application sends a stop message to Bi-i. The 'ibis.set' instructions are not needed, because the 'ibis.init' instruction sets all of the Ibis parameters to their default values.

2.14. ACE16k

ACE16k (Focal Plain Array Processor) is connected directly to the DSP, therefore the images from it can be obtained directly from the DSP. The whole memory of the ACE16k can be reached from AMC by a predefined symbol.

```
Analog gray scale image memories: LAM1 - LAM8
Binary image memories: LLM1 - LLM2
Template memories: TEM1 - TEM32
```

These containers don't map to the DSP memory ranges, therefore there are some transfer instructions in AMC to enable the transfer between them.

2.14.1. Image And Template I/O

We can write and read templates and images to/from the ACE16k.

Example 2.15. Image I/O for ACE16k in AMC

```
;Initializes ACE16k size images on DSP memory.
conv.glm.llm FALSE LLM3
conv.gam_b.lam 100 LAM9

;Writes images to ACE16k
mov.llm.llm LLM3 LLM1
mov.lam.lam LAM9 LAM1

;Reads images from ACE16k
mov.llm.llm LLM1 LLM4
mov.lam.lam LAM1 LAM10
```

Important

Keep in mind that the image memories of ACE16k only keep their contents for a few 10 milliseconds.

Example 2.16. Template I/O for ACE16k in AMC

Before writing a template to the ACE16k it has to be loaded from the host computer to the DSP memory.

```
;Load template from the host computer.
host.load.tem erosion.tem TEM33

;Write template to the ACE16k
mov.tem.tem TEM33 TEM1

;Read template from ACE16k
;mov.tem.tem TEM1 TEM34
```

Note

The read back template values can be different from the original template values because the floating-point template values will be converted to unsigned integer value (0 - 255) on the ACE16k.

2.14.2. Logical Operation

In bit-wise logic operations on the ACE16k each operand must be a binary image memory, however the output may be either a binary or a gray scale image memory on ACE16k.

Example 2.17. Bit-wise logical operations on ACE16k in AMC

```
;Writes the previously loaded binary images to AE16K
mov.llm.llm LLM3 LLM1
mov.llm.llm LLM4 LLM2

;Bit-wise logical OR operation on ACE16k to binary image memory,
ar.or.llm LLM1 LLM2 LLM2
;or to gray scale image memory.
ar.or.llm LLM1 LLM2 LAM1

;Reads results from ACE16k
mov.llm.llm LLM1 LLM5
mov.lam.lam LAM1 LAM9
```

2.14.3. Template Execution

Template execution (transient evolution) is the chief operation of ACE16k. The image processing library is also based on this operation. There are two modes of this, namely continuous transient evolution which runs for a specified time and stops, and the iterative discrete time evolution which runs for a unit time in all iterations and the output of an iteration will be the input and state of next iteration (useful for morphological operations). There are two masks for evolution, namely writing mask and the freezing mask which affect the individual pixels during evolution.

Example 2.18. Continuous template execution in AMC

```
Write images and template to ACE16k.
mov.tem.tem tem33 TEM1
mov.llm.llm llm3 LLM1

;Executes TEM1 template on LLM1 in iterative mode in 5 iteration
ar.tem TEM1 LLM1 LLM1 LAM4 5 1 1 NIL NIL NIL HW_MODE_DISC

;Reads result image from ACE16k
mov.lam.lam LAM4 LAM9
```

Example 2.19. Iterative template execution with freezing mask in AMC

```
Write images and template to ACE16k.
mov.tem.tem TEM33 TEM1
;Writes input image to ACE16k
mov.lam.lam LAM9 LAM4
;Write mask image to ACE16k
mov.llm.llm LLM3 LLM1

;Execute template with Freezing Mask
ar.tem TEM1 LAM4 LAM4 LAM4 2.0 0 -1 LLM1 NIL NIL HW_MODE_CONT

;Reads result image from ACE16k
mov.lam.lam LAM4 LAM10
```

2.14.4. Image Acquisition

There is a CMOS sensor above the array processor on the ACE16k which can acquire gray scale images to its image memories.

Example 2.20. Image acquisition with ACE16k in AMC

```
;Acquisition to gray scale image memory with 500 micro sec integration time.
mov.opt.lam LAM1 500 195

;Acquisition to binary image memory with 500 micro sec integration time.
mov.opt.llm LLM1 500 195
```

If the target is a binary image memory, the acquired image will be cut off with 128 (middle of the pixel value range). Specifying the Precharge value is not needed, because it has default value and using this default value is highly recommended. If it is modified the contrast will be changed.

2.15. Image Processing Library (IPL)

The IPL has two main groups of operations, morphological and gray scale operations which run on the ACE16k (instruction names are vmp.xxx) and it has a totally separate group which runs on DSP (instruction names are dsp.xxx). If you want to use the first two groups, an initialization step (vmp.init instruction) is needed, which will overwrite the first 21 template memories with the necessary IPL templates. If the morphological group is used a calibration step (vmp.start instruction) is needed for correct operation once every 10 milliseconds.

Example 2.21. Typical morphological procedure with IPL in AMC

```
;IPL initialization.
vmp.init vmp_morph

;Write the previously loaded image to ACE16k
mov.llm.llm LLM3 LLM2
;Calibration
vmp.start
;For example an erosion
vmp.morph.erode4 1

;Read the result image from ACE16k
mov.llm.llm LLM2 LLM4
```

Example 2.22. Typical gray scale procedure with IPL in AMC

```
;IPL initialization.
vmp.init vmp_gray

;Write the previously loaded image to ACE16k
mov.lam.lam LAM9 LAM1
;For example a horizontal sobel filtering
vmp.gray.sobel LAM1 LAM2 sobel_horizontal

;Read the result image from ACE16k
mov.lam.lam LAM2 LAM10
```

2.16. Profiling

Profiling means the measurement of the execution time of operations on the DSP. We can measure the running time of a given part of AMC algorithm with this method. For example the working time of a function from the image processing library:

Example 2.23. Measuring time in AMC

```
;Initializes and starts the timer.
timer.start

vmp.morph.erode4 1

;Write the elapsed time to the GAM1 scalar address.
timer.read GAM1
```

The unit of the obtained time value in GAM1 depends on the Bi-i version, for example if the device is a Bi-iv2 (600MHz) and the value of GAM1 is 750, the effective time of erosion is:

Time = (1/600) * 8 * 750 = 10цs.

If the effective time is needed, it can be calculated with scalar operations:

```
;GAM1 = 750

sc.mul.gam 8 GAM1 GAM1
sc.div.gamf GAM1 600 GAMF1
```

The value of GAMF1 is a floating-point number which is the running time in microseconds.

3. AMCedit

AMCedit is a graphical environment for developing and running AMC (Analogic Macro Code) programs on Bi-i systems. It is a part of all Bi-i software packages. The AMC source file contains AMC statements and it is compiled to ABC (Analogic Binary Code) file. *AMCedit* invokes *IVRun_x86* to execute the compiled ABC.

3.1. Editing AMC codes

The user interface of *AMCedit* has two main parts: the editor and the info-tree window. The editor window is a standard text editor field. You can write your code here. The info-tree window is an information tree. You can find here various information about the AMC syntax and your AMC code.

3.1.1. The Editor Window

AMCedit provides several features that can make the AMC development easy:

- Syntax coloring
- Automatic completion
- AMC Syntax dialog
- Info-tree and its usage by mouse

3.1.1.1. The AMC Syntax Dialog

The AMC Syntax dialog helps finding the proper AMC instruction. The AMC Syntax item of the Help menu or the toolbar icon to invoke the AMC Syntax dialog.

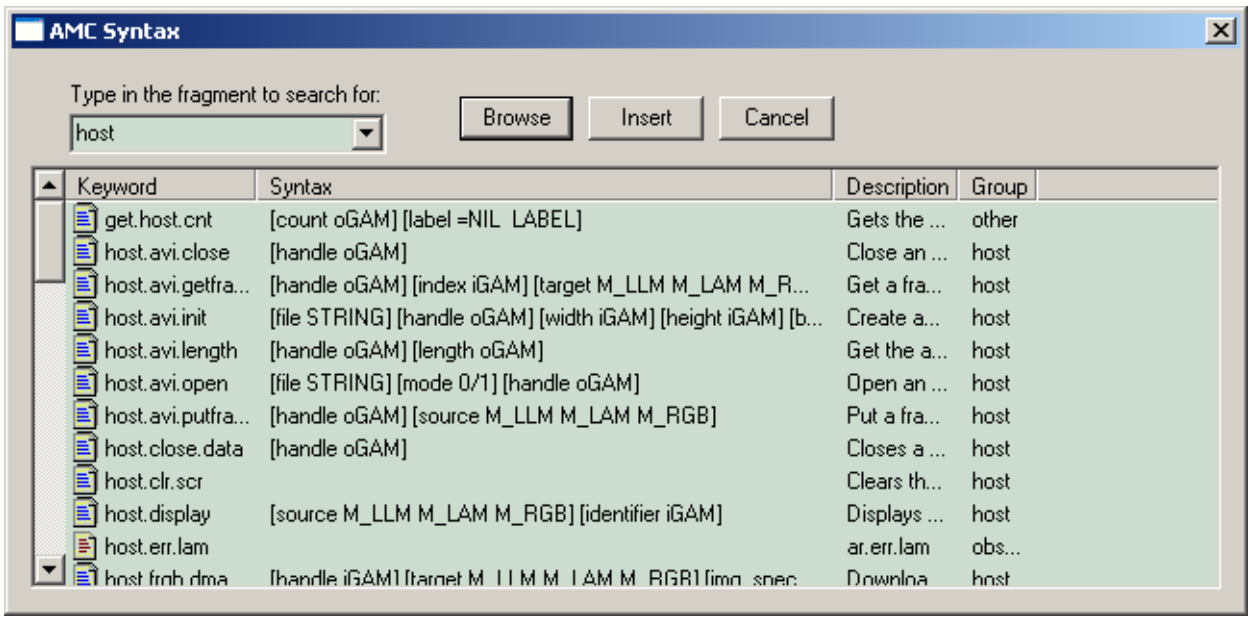

Figure 3.1. AMC Syntax dialog

Dialog options

Type in the fragment to search for: Write a new search term here or select from the previous ones.

Browse button Starts searching.

Insert button Exits and inserts the selected keyword.

Fields of the result window

Keyword The name of the keyword.

Syntax The exact syntax description of the keyword. [parameter name = default value TYPE]

Description The description of the statement.

Group The name of the keyword group.

3.1.1.2. Auto Complete

Select the Auto Complete item of the View menu to switch on/off the automatic completion feature. Use Ctrl-Space to complete the currently edited word to a valid AMC instruction. It works only if the completion of the current segment is unambiguous.

3.1.2. The Info-tree Window

The info-tree contains several information about the current document and also about the syntax of the AMC language. The result of the compilation can also be found here. Here is a brief description about the items of this tree:

Language syntax Contains the syntax of the AMC language. You can find here the instructions, argument types, etc. in several groups. In the Status Bar you can find the brief description of the selected language element.

Insert Text In this item there are some predefined code segments

Labels The labels in the current document are collected here.

Templates The templates in the current document are collected here.

Defines The definitions in the current document are collected here.

Images The images in the current document are collected here.

Compile info Here you can find the result of the compilation.

The effect of different mouse manipulations on various parts of the info-tree is shown in the table below.

Info-tree item type	Left double click	Right double click
Language syntax	Search the item in your code	Insert the item into your code
Insert text	Insert the function corresponding to the item	
Images	Launch your image editor	Insert the line into your code
Defines	Jump to the definition	Insert the defined text
Labels	Jump to the label	Insert the label name
Templates	Open the template file	Insert the line into your code
Compile info	Insert the info into your code as a comment	
Compiler file	Open the file in the editor	Insert the file name
Compile error	Jump to the error line	Insert as a comment

Table 3.1. Operations with the mouse on the info-tree

3.1.3. Status Bar

The status bar has the following fields:

- Help text
- Line number if the focus is on the editor field
- Caps lock status
- Num Lock status

The help text contains a short description of the current menu item or the toolbar button, when you navigate through the menu or the toolbar. This area also shows messages, when you are editing code or browsing in the info-tree. These message refers to the syntax of the currently edited command or the properties of the current info-tree item.

3.2. Running the AMC

Before running your code you must have a valid device created with *IVConfig* that points to your Bi-i. When you run a code *AMCedit* compiles the AMC file to ABC file, and then starts the *IVRun (IVRun_x86.exe)* software to execute this ABC code on the selected device.

To execute an AMC code follow these steps:

- Select the target device.
- Select an *IVRun* Workspace File for your AMC code (optional).
- Compile the code.
- Run the code.

3.2.1. Selecting the Device

Use the Device item of the Project menu or the toolbar icon to invoke the Select Device dialog.

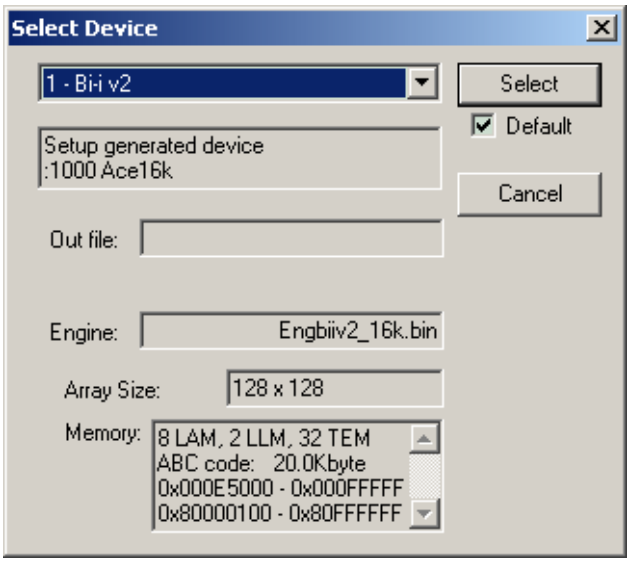

Figure 3.2. The Select Device dialog

Device Here you can select the Bi-i device that will be used for program execution. The device must be configured with I*VConfig* prior to running *AMCedit*. The device must have the following properties:

- The device type must be Bi-i v2.
- The *Use AMC/ABC* check box must be checked.

The setting above are defaults, newly created devices always have these features. Under the combo box some information about the selected device can be found.

Default checkbox Save the selected device as default. See menu Project/Preferences.

3.2.2. Selecting the Workspace

Select the Options item of the Project menu to specify a *IVRun* Workspace file, which will be passed to *IVRun* when you execute the AMC code. This workspace must refer to the ABC file whose source is the current document. If you do not set any workspace, then *AMCedit* will find a workspace named like the AMC file, but if it does not exist then executes the ABC file without any workspace.

3.2.3. Compiling the AMC

To compile the AMC file use the Build item of the Project menu or toolbar icon. The result of the compilation with error info can be found at the bottom of the info-tree.

3.2.4. Running the ABC

To run the compiled code select the Execute item of the Project menu or on the toolbar. This command starts *IVRun* to execute the ABC code on the selected device. The code will be compiled before running, if it has changed after the last compilation.

3.3. Menu Items, Toolbar Buttons and Shortcut Keys

- File menu
  - New: Creates a new document in *AMCedit*

    Toolbar:

    Shortcut: Ctrl-N
  - Open: Opens an existing document in a new window

    Toolbar:

    Shortcut: Ctrl-O
  - Close: Closes the active window
  - Save: Saves the current document

    Toolbar:

    Shortcut: Ctrl-S
  - Save As: Saves the document on a new name

    Shortcut: F2
  - Print: Prints the current document

    Shortcut: Ctrl-P

- Print Preview: Displays the active document as it would appear when printed

  Toolbar:

- Print Setup: Displays the Print Setup dialog box where you can specify the printer and print properties
- Recent Files: List of the recently used files
- Exit: Exits from *AMCedit*

  Shortcut: Alt-F4

- Edit menu
  - Undo: Reverses the last few editing actions if possible

    Toolbar:

    Shortcut: Ctrl+Z
  - Redo: Reverse the last undo actions if possible

    Shortcut: Ctrl+Y
  - Cut: Remove the currently selected data from the document and puts it on the clipboard

    Toolbar:

    Shortcut: Ctrl+X
  - Copy: Copies selected data onto the clipboard

    Toolbar:

    Shortcut: Ctrl+C
  - Paste: Inserts a copy of the clipboard contents at the insertion point

    Toolbar:

    Shortcut: Ctrl+V
  - Clear: Removes the currently selected data from the document

    Shortcut: Del
  - Delete Line: Removes the currently edited line from the document

    Shortcut: Ctrl+K/E
  - Select All: Selects all text in the active window or selects all text in the selected object

    Shortcut: Ctrl+A
  - Find: Searches for the specified text, formatting symbols, comments, footnotes or endnotes in the active document

    Toolbar:

    Shortcut: Ctrl+F

- Replace: Searches for the specified text and replaces it in the active document

  Shortcut: Ctrl+H

- Repeat: Repeats the last search

  Shortcut: F3

- Go To Line: Moves the insertion point to head of the line you want to go

  Shortcut: Ctrl+G

- View menu
  - Toolbar: Displays or hides the Toolbar
  - Status Bar: Displays or hides the Status Bar
  - Move Split: Moves the line between the tree and the editor windows
  - Swap Focus: Changes the focus between the tree and the editor window

    Shortcut: F4

  - Change Layout: Saves the file and changes the layout between vertical and horizontal orientation

    Toolbar:

  - Auto Complete: Completes the current statement

    Shortcut: Ctrl+Space

  - Set Font: Selects the font properties
- Insert menu
  - Filename: Inserts a file name
  - File Content: Inserts the content of a file into your document

    Toolbar:

  - Block Define: Inserts '#define ' at the insertion point

    Shortcut: Ctrl+#

  - Block Comment: Comments the selected lines

    Toolbar:

    Shortcut: Ctrl+R/D

  - Block Uncomment: Removes the comment symbols form the selected lines

    Toolbar:

    Shortcut: Ctrl+T/Shift D

  - Make Lowercase: Makes lowercase the selection
- Project menu

- Device: You can select the target device here

  Toolbar:

- Options: You can select here a *IVRun* Workspace File which will be taken to the *IVRun* when you execute the AMC code

- Build: Builds the current document

  Toolbar:

  Shortcut: Ctrl+F9; F7

- Execute: Compiles and executes the current document

  Toolbar:

  Shortcut: F9; F5

- Preferences: Displays the Remember dialog box where you can specify some editor and project settings for the next startup

- Window menu
  - Cascade: Arranges multiple opened windows in an overlapped fashion
  - Tile Horizontal: Arranges multiple opened windows vertically in a non-overlapped fashion
  - Tile Vertical: Arranges multiple opened windows side by side in a non-overlapped fashion
  - Next Window: Switches to the next open document window

    Shortcut: F6

  - List of the currently opened documents
- Help menu
  - Help: Displays the *AMCedit* online help

    Shortcut: F1

  - AMC Manual: Displays the AMC Manual online help
  - AMC Reference: Displays the AMC Reference online help
  - AMC Syntax: Displays the AMC Syntax dialog box

    Toolbar:

    Shortcut: Ctrl-F1

  - About *AMCedit*: Displays the About dialog

4. The API

The API is a programming interface to be used on the host computer for writing applications working together with Bi-i. For simplicity, the application that runs on the host computer accessing the API is referred to as the host application. The host application can run programs on the Bi-i and exchange data between the Bi-i and the host computer. The default host application is *IVRun (IVRun_x86.exe)*, which is installed with all software packages. Use the API if you need a host application beyond the scope of *IVRun*. The API was developed for Bi-i systems. For example, if you need a customized user interface or if your application has to do other tasks besides working with Bi-i (such as accessing a database or different peripherals).

The API can executes DLL based applications like IVRun. It is very useful if you would like develop the algorithm and user interface on PC without Bi-i system. The API interface is same for the Bii and DLL based applications. The See the InstantVision ISE User's Guide Chapter 3 for more details from DLL based applications.

API is implemented as a Windows dynamic library called *HostAPI_x86.dll*. It can be used with practically any development environment that can work with dynamic libraries. The prototypes of all functions and all the definitions are located in a single C++ header file called *API.h*. The import library called *HostApi_msvc7_x86.lib* can be used for integrating your application with API under Microsoft Visual C++ in this case, you only need to pass *HostApi_msvc7_x86.lib* to the linker. Consult the documentation of your development environment on using dynamic libraries if you're working with a different development environment.

API does not control the program execution on the Bi-i directly. The basic idea is that two independent applications are running: one inside the Bi-i and another on the host computer. They can send messages to each other and they can exchange data, but they are not linked closely.

4.1. Running an Application

Before running an application with the API, configure a device with *IVConfig* that points to your Bi-i or a Local device and (optionally) contains the DSP program to be executed on the Bi-i or the name fo the application DLL.

There are several ways of receiving messages from the application running. You must decide which mode you choose before running the application. It cannot be changed later. The messaging modes are listed in the *TInfoType* enumeration as follows:

BII_INFO_THREAD Messages from the Bi-i application will be sent to a thread.

BII_INFO_WINDOW Messages from the Bi-i application will be sent to a window.

BII_INFO_CALLBACK A callback function will be called when a message from the Bi-i is received. The function must match the *Bii_Callback* type declaration.

BII_INFO_QUEUE Messages from the Bi-i application are sent into an internal queue that can be checked in the host application.

After configuring a device and deciding the messaging mode, the simplest way of running an application on the Bi-i is calling the *Bii_Run* function with the following parameters:

Device The number of the device as it was set with *IVConfig*.

App The name of the application. Pass NULL or empty string, if you want to run the default application that was set in *IVConfig*. If given without a full path, it is searched relative to the *bin* folder within the installation directory of the Bi-i software.

BinFile The name of the ABC file. Only relevant when using AMC, pass NULL or empty string, when running an application created with the Bi-i SDK. It must be given with a fully qualified path.

DefDir The default data directory. For all host operations on the Bi-i (AMC or SDK), the files are located relative to this directory, unless they are given with full path. Pass NULL or empty string, if you do not want to use this feature.

InfoType Refers to the mode of message handling as it was listed above.

InfoParam The parameter for message handling. This is a union, always fill only the field that is relevant in your case. See the table below.

InfoType	InfoParam field	Meaning
BII_INFO_THREAD	ThreadId	The identifier of the thread to which the messages are posted
BII_INFO_WINDOW	WindowHandle	The handle of the window to which the messages are posted
BII_INFO_CALLBACK	Callback	The pointer to the callback function.
BII_INFO_QUEUE	Ignored	

Table 4.1. InfoParam to be passed depending on InfoType

The return value of *Bii_Run* is a handle that can be used later with several other functions.

A more complicated approach is the following sequence:

- Call the *Bii_Init* function, which has exactly the same parameters as *Bii_Run*. The only difference is that it only downloads the application to the Bi-i, but it does not start it.
- Call the *Bii_Start* function to start the application.
- Call the *Bii_Close* function to close the connection with the Bi-i.

Using this method can be useful when you want to run the same application on the Bi-i several times without downloading it each time. Another advantage can be that you can prepare the execution with *Bii_Init*, and after the application can be started in a relatively short time with *Bii_Start*.

The messages between the host application and the Bi-i have the following components:

Identifier A value which identifies the role of the message.

Parameter Specifies additional information about the message. The exact meaning depends on the value of the Identifier field.

Additional parameter Always contains the handle (the return value of Bii_Run or Bii_Init), when receiving messages from the Bi-i.

Receiving the handle as an additional parameter is useful, when more Bi-i systems are used from the same host application at the same time. Otherwise this parameter can be omitted. When sending messages to the Bi-i, the Parameter and Additional parameter components are currently not used, they are reserved for future use. Passing zero is recommended.

The example below calls *Bii_Run* for device 1, and the current thread will receive the messages.

Example 4.1. Calling Bii_Run

```
Bii_InfoParam infoparam;
void *handle;

infoparam.ThreadId=GetCurrentThreadId();//Getting the Id of the current thread
handle = Bii_Run(
  1,    //The number of the device in IvConfig
  NULL, //The default DSP program from IvConfig
  NULL, //No ABC file
  NULL, //No default directory
  BII_INFO_THREAD, //The current thread will receive messages
  infoparam
  );
```

4.2. Receiving Messages

When using the BII_INFO_THREAD or BII_INFO_WINDOW mode, messages can be received with the *GetMessage* or the *PeekMessage* Windows API calls. Both functions receive the message in an *MSG* structure. The identifier, parameter and additional parameter components are placed into the 'message', 'wParam' and 'lParam' members respectively.

When using BII_INFO_CALLBACK mode, then all message components are passed to the callback function.

When using BII_INFO_QUEUE mode, then the internal message queue can be checked with the *Bii_PeekMessage* and the *Bii_GetMessage* functions. The difference between them is that *Bii_PeekMessage* returns immediately, while *Bii_GetMessage* waits for a message. Both receive the identifier and the parameter only. The additional parameter is not received, because the handle that would be received as additional parameter is passed in the first argument of the functions. When working with more Bi-i systems at the same time, then the message queue for each of them must be checked in separate function calls.

The host application can receive the following messages from the Bi-i:

BII_MSG_START — Sent after calling *Bii_Run*, when the application has started on the Bi-i. Not sent when the application is started with *Bii_Start* or if there was an error before starting the application, for example, the Bi-i was not available on the network. BII_MSG_START has no parameter.

BII_MSG_PUT — The Bi-i sends data to the host application. See Section 4.4, “Data Transfer” for more details.

BII_MSG_CLR — The host application should clear its internal data. See Section 4.4, “Data Transfer” for more details.

BII_MSG_GET — The Bi-i asks for data from the host application. See Section 4.4, “Data Transfer” for more details.

BII_MSG_RELEASE — Used in conjunction with BII_MSG_GET. See Section 4.4, “Data Transfer” for more details.

BII_MSG_WAIT_BEGIN — Indicates that the Bi-i application enter a wait state, which is a fixed amount of delay. The length of the wait state in ms is passed as parameter.

BII_MSG_WAIT_END — Indicates the wait state is finished on the Bi-i. This message has no parameter.

BII_MSG_STOP — Sent when the application on the Bi-i stopped. BII_MSG_STOP has no parameter. This invalidates the handle that was received as the return value of *Bii_Run*. Do not use the handle after receiving this message.

The example below demonstrates how messages are received in a thread:

Example 4.2. Receiving messages in a thread

```
MSG msg;

while (go_ahead)
  {
  GetMessage(&msg,NULL,0,0);
  switch (msg.message)
    {
    case BII_MSG_START:
    //Application started
    break;
   case BII_MSG_STOP:
    //Application stopped
    go_ahead = false;
   break;
   ....
    }
   }
```

4.3. Sending Messages

Use the *Bii_PostMessage* function the send the following messages to the Bi-i:

BII_MSG_PAUSE	The Bi-i pauses program execution.
BII_MSG_RESUME	The Bi-i resumes program execution, when it was temporarily paused with BII_MSG_PAUSE.
BII_MSG_WAIT_END	Use when BII_MSG_WAIT_BEGIN was received. Forces the Bi-i application to finish wait state immediately without waiting for the end of the specified delay.
BII_MSG_STOP	Stops the execution of the program on the Bi-i. The Bi-i sends back the same message, when the program was really stopped.
BII_MSG_ABORT	Aborts the program on the Bi-i. Use this message with care, because the application has no chance of executing any cleanup code. The normal way of stopping the application is the BII_MSG_STOP message. Do not use the handle received from *Bii_Run* after sending this message.

4.4. Data Transfer

4.4.1. Data Representation

The data transfer between the Bi-i and the host application is based on the *Bii_HostData* structure. The data representation on the Bi-i uses the data types defined in the *InstantVision* library. Consequently, some elements of *Bii_HostData* also use these data types. See the *InstantVision* documentation for more details.

The *Bii_HostData* structure has the following members:

Type	The type of the data, selected from the *InstantVision::TDataType* enumeration.
Identifier	An integer that identifies the data. In *IVRun*, this is the number of the window (1,2,..). In your application you can use this number for any purpose.
Anonymous union	A union that actually contains the data in various fields depending on the *Type* member. See the table below.

Type	Union member	Representation
DATA_TYPE_BIT	bData	unsigned char
DATA_TYPE_BYTE		
DATA_TYPE_SHORT	sData	short
DATA_TYPE_INT	Data	unsigned int
DATA_TYPE_FLOAT	fData	float
DATA_TYPE_xxx_MATRIX	DataPtr	InstantVision::TMatrixHeader*
DATA_TYPE_RGB		
DATA_TYPE_IMAGE	Dib	void*
DATA_TYPE_TEXT	StrPtr	char*

Table 4.2. The members of the union in the Bii_HostData structure

DATA_TYPE_IMAGE is stored as a device-independent bitmap as it is described in the Microsoft Windows documentation. DATA_TYPE_TEXT is a null-terminated string.

Use the *Bii_AllocHostData* function to allocate a new host data structure for a specific data type. The required spaces for the given dimensions is allocated behind the data pointer within the same function call. The reverse function is *Bii_FreeHostData*, which can free the Bii_HostData structure including the area behind the data pointer.

4.4.2. Data Transfer Messages

Data from the Bi-i is received in the host application with the *BII_MSG_PUT* message. The parameter of the message is a pointer to the *Bii_HostData* structure. When receiving this message, call the *Bii_SetDataEvent* function showing that the host request has been processed. You can free the allocated memory with *Bii_FreeHostData*, if you do not need the received data any more.

The *BII_MSG_CLR* message is sent, if the Bi-i application wants to clear the data stored on the host side. This message has no parameter. The practical implementation depends on your application. *IVRun* closes all windows, when it receives this message. When processing BII_MSG_CLR, the *Bii_SetDataEvent* function must be called.

The *BII_MSG_GET* message is sent to the host application, when it should send data to the Bi-i. The parameter of this message is a pointer to the *Bii_HostDataRequest* structure, which consists of an Identifier and a Type field, and a pointer to the *Bii_HostData* structure. The host application must set this pointer to a valid value, if it can supply the requested data, or to NULL, if it cannot supply it. The data supplied must match the data requested:

- The Identifier must be the same in the *Bii_HostData* structure as it is in the *Bii_HostDataRequest* structure.
- The Type in the *Bii_HostData* structure must be compatible with the Type the *Bii_HostDataRequest* structure. Matching data types are obviously always compatible. Besides that *DATA_TYPE_BIT_MATRIX*, *DATA_TYPE_BYTE_MATRIX* or *DATA_TYPE_RGB* can be sent if *DATA_TYPE_IMAGE* was requested. The opposite direction is somewhat restricted: *DATA_TYPE_IMAGE* with the matching color depth can be sent when *DATA_TYPE_BIT_MATRIX*, *DATA_TYPE_BYTE_MATRIX* or *DATA_TYPE_RGB* was requested.

When sending dynamic data, the matching type of data structure must exist in the memory of the Bi-i. In the Bi-i, all dynamic data types are represented as an object derived from *InstantVision::TAncestorMatrix*. *DATA_TYPE_IMAGE* is always converted to *TBitMatrix*, *TByteMatrix* or *TRGBMatrix*, depending on the color depth. *DATA_TYPE_TEXT* is converted to *TString*. The target object will be resized, if its size is different than it was sent. *TString* is an exception: it will be resized only if the string sent is longer than the current size of the object.

After setting the pointer in the *Bii_HostData* structure, the *Bii_SetDataEvent* function must be called, which indicates, that the request has been processed.

The *BII_MSG_RELEASE* message is always received, when data was sent to the Bi-i in conjunction with the *BII_MSG_GET* message. The parameter of this message is a pointer to a *Bii_HostData* structure, practically the

same as it was set within the *Bii_HostDataRequest* structure. The role of this message is denoting that the requested data is not needed any more on the host side, because it is already physically in the Bi-i. The host application may want to free the memory, if it was allocated, when the *BII_MSG_GET* message was processed. If it is not the case, you can omit this message.

4.4.3. Direct Transfer

The host application can directly read and write the memory of the DSP, even if no messages from the Bi-i were received.

Call the *Bii_WriteData* function to write the Bi-i memory. A physical address and a *Bii_HostData* structure are passed to this function. When writing data with a dynamic size, the matching type of object derived from *TAncestorMatrix* with exactly the same size must exist in the Bi-i memory at the given location prior to calling *Bii_WriteData*. *DATA_TYPE_IMAGE* can be written to *TBitMatrix*, *TByteMatrix* or to *TRGBMatrix*, depending on the color depth. *DATA_TYPE_TEXT* is converted to *TString*, which need not have exactly the same size: the target size can be bigger.

Call the *Bii_ReadData* function to read the Bi-i memory. A physical address and a *Bii_HostDataRequest* structure are passed to this function. The Identifier filed of the structure is ignored in this case. The HostData field is filled with the requested data matching the type passed. When reading data with a dynamic size, the matching type of object derived from *TAncestorMatrix* must exist in the Bi-i memory at the given location prior to calling *Bii_ReadData*. *DATA_TYPE_IMAGE* reads any of *TBitMatrix*, *TByteMatrix* or *TRGBMatrix*, the required conversion is done internally. Similarly *DATA_TYPE_TEXT* is converted from *TString*. The memory must be deallocated later with *Bii_FreeHostData*.

4.5. Cooperation with AMC and SDK

The table below shows the functions in AMC and SDK that results in sending message to the host application.

Message	AMC	SDK
BII_MSG_PUT	host.display, host.print, host.put.data	Operator TStd<<
BII_MSG_CLR	host.clr.scr	TStd::ClrScr()
BII_MSG_GET	host.get.data or host.get.pic	Operator TStd>> in standard input mode
BII_MSG_WAIT_BEGIN	host.wait	TStd::Wait()
BII_MSG_WAIT_END		
BII_MSG_STOP	end or BII_MSG_STOP received from the host	Return from main() or Program_End ()

Table 4.3. Messages received in the host application

The table below show the effect of messages that where sent to the Bi-i in AMC and SDK.

Message	AMC	SDK
BII_MSG_PAUSE	Pause after the current instruction	GetPauseStatus() or CheckPauseStatus() indicates it
BII_MSG_RESUME	Resume after pause	
BII_MSG_WAIT_END	End of wait state immediately	
BII_MSG_STOP	Stop ofter the current instruction	GetStopStatus() indicates it
BII_MSG_ABORT	Abort immediately	

Table 4.4. Messages sent from the host application

4.6. .NET Components

This is a component package for the Microsoft Visual Studio 2005 that works like an API for the .NET framework. There are five components in this package:

- **DeviceControl** which is an invisible component that presents the interface of the Bi-i API to the .NET framework. It contains all of the properties, events and public functions, that are necessary to access and control an InstantVision application.

  It can start, stop, pause and resume an application and gives the control interface of the API by events. Two types of events are used:

  - Control events. These are triggered if a control event occurred (start, stop, wait, etc.).
  - Data events. These occur if the device sends or requests data (IntReceived, ImgRequested, etc.)

- **DeviceSelector** which is a combo box for selecting the device that was created by the IVConfig. The device parameters are read from the Registry.

  The result list can be filtered by the type of the devices.

- **ExecutionControl** which is a four-button panel used to control the execution.

  It contains a Start, a Stop, a Pause and a Resume button, clicking on which triggers the relevant action.

- **ParamXMLHandler** which is an invisible component, that handles the parameters. It parses the parameter description XML file and creates a description structure. It validates the parameters and uses the device control to send the new parameter value to the device. If there is any error during the parameter modification, it throws a TParamException to inform the user about it.

- **ParamGrid** is a custom property grid that makes the parameter visible and easily modifiable.

Note

The .NET components will not work with Visual Studio 2003 .NET or earlier versions.

4.6.1. Installing the Components

- Install components to the Global Assembly Cache
  - Open the *Control Panel* of the computer
  - Select the *Administrative Tools*
  - Start the *Microsoft .NET Framework 2.0 Configuration* (.NET Framework 2.0 SDK needed)
  - Open the *MyComputer* in the tree view on the left side.
  - Right click on the *Assembly Cache* and select *Add*
  - Select the **IvIseDotNet_x86.dll** from the **bin** folder of the **InstantVisionISE**
  - Close the *Microsoft .NET Framework 2.0 Configuration*
- Install components to the Microsoft Visual Studio environment:
  - Start the Microsoft Visual Studio 2005 (VS 2003 will not work)
  - Select the *Options...* from the *Tools* menu
  - Open the *Projects and Solutions* in the tree view

    - Select *VC++ Directories*
    - Add the **bin** folder of the **InstantVisionISE** to the *Reference files* and the *Executable files* on Win32 platform.
    - Add the **include** folder of the **InstantVisionISE** to the *Include files* on Win32 platform.
    - Add the **lib** folder of the **InstantVisionISE** to the *Library files* on Win32 platform.
    - Click the *OK* button and the components are ready to use
- Install components within a Project:
    - Start the Microsoft Visual Studio 2005
    - Create a new *Windows Forms Application* project
    - Make the *Toolbox* visible from the view menu
    - Select the *Choose Items...* from the pop up menu of the *Toolbox*
    - Click on the *Browse...* button
    - Select the **IvIseDotNet_x86.dll** from the **bin** folder of the **InstantVisionISE**
    - Click the *OK* button and the components are ready to use
- Required Environment settings:
    - Add the **bin** folder of the **InstantVisionISE** to the *System Path*.

4.6.2. Usage of the Components

This is a sample that shows the components and their usage in a very simple application. The sample can be found in the samples folder under the **InstantVision ISE** (samples/API/DotNetComponents). Only the DeviceControl component is essential, because it is the interface of the API. The other components only improve usability.

- Place the necessary components on the application form.

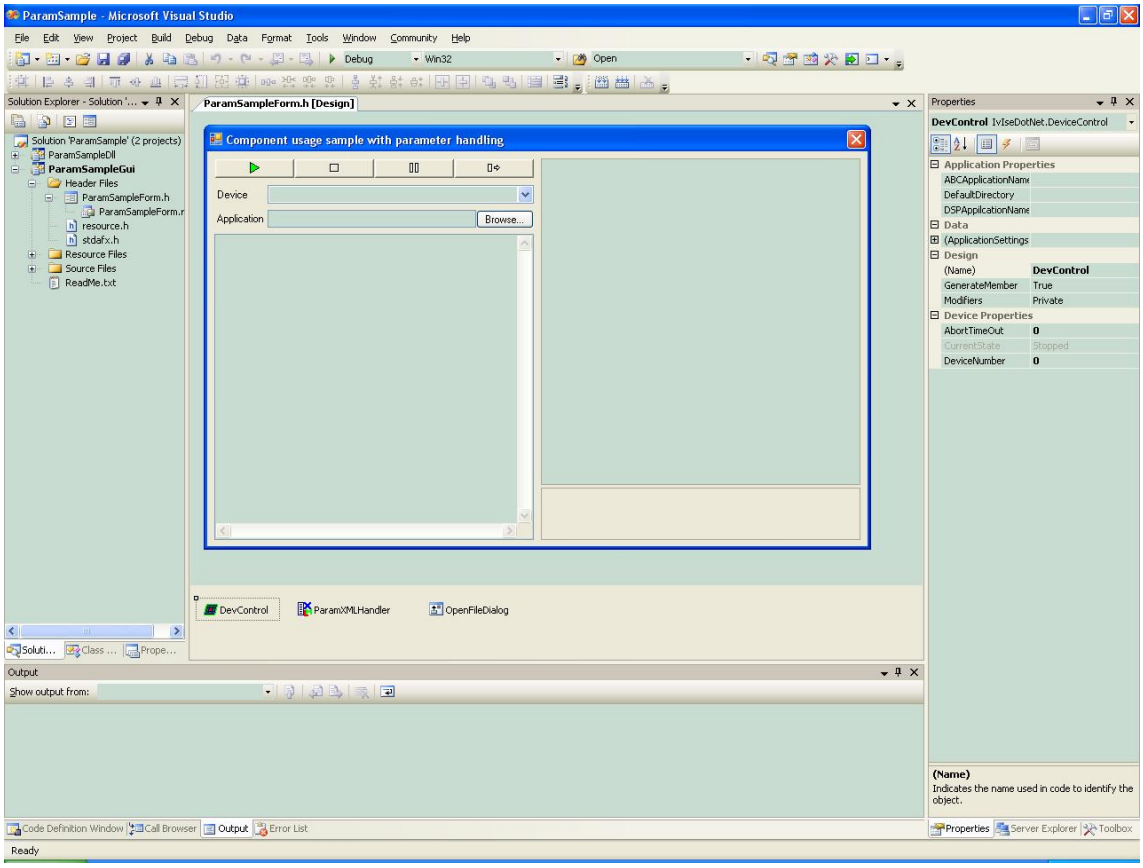

Figure 4.1. Add the components

- Link the components together

  The DeviceSelector, the ExecutionControl and ParamXMLHandler components modify or use the DeviceControl's properties, therefore it is necessary to link these components to the DeviceControl. Use the DeviceControler property to link the components. The ParamGrid uses the ParamXMLHandler so they also should be linked. Use the parameter handler property to link the components.

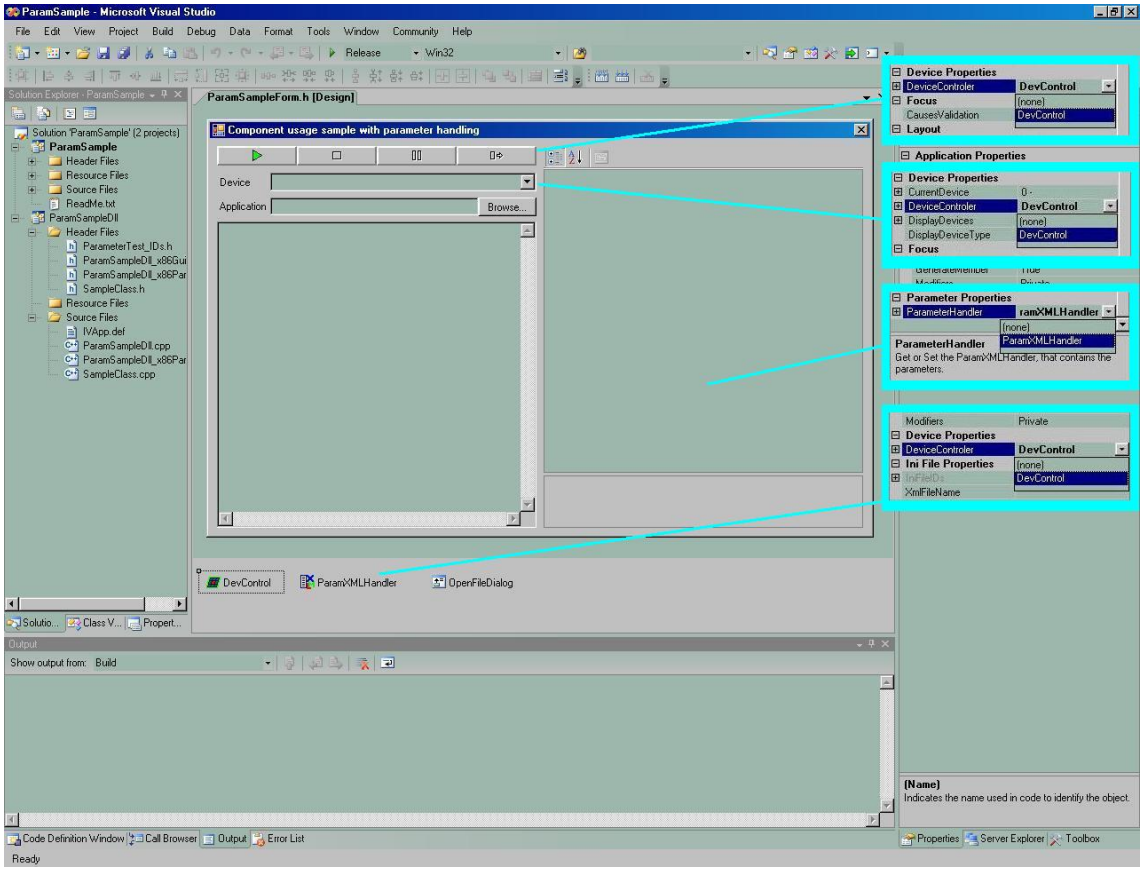

Figure 4.2. Link the Components

If the components are linked the application is ready to start and the data events will occur if the device side code triggers the transaction.

4.7. Parameter generation (ParamTool utility)

This is the application called 'ParamTool'. It can be found in the **bin** folder of the I**nstantVision ISE**. This application is responsible for generating the parameter description XML and the device side source code of the parameter handling class. The application works with the header files of the application, that have to be commented specially. Primarily, this utility is developed to prepare the interface of the parameter handler components (ParamXML-Handler, ParamGrid), but if there is **no GUI** for the application it can be useful, because it has the capability to handle the parameters off-line. For more details, see Section 4.7.2, “Parameter Handling Class”.

The parameters are those member variables of the application classes that can be modified by the user interface during execution. These parameters are stored in standard ini files. Each parameter consists of an ini file and a section. It also has a name within the section. These three things make up the unique identifier of a parameter.

To make a parameter from a member variable a special comment block should be placed before the variable and the container class. The blocks should be started with a slash and two asterisks and should have the structure as shown in the following example.

Example 4.3. Sample Parameter Comment block

```
/**
  This is an sample class.
  @ClassPath SampleNamespace::SampleClass
*/
class SampleClass {

/**
  @IniFile  MainParamFile
  This is the main parameter file that contains the
  main parameters of the application.

  @Section MainSection
  This section contains the parameter of the Main cycle.
*/

public:
  /** This is a sample integer variable.

    @IniFile  MainParamFile
    @Section  MainSection
    @Key      SampleInt
    @Datatype DATA_TYPE_INT
    @Range    [-450000,480000]
    @DefaultValue 12000
  */
  int  SmpInt;
```

4.7.1. Special comments

The comment block is started with the description of the parameter. No special keyword is needed to specify it. The ini file's or the section's description can be given after the @IniFile or the @Section keyword, if no @Key keyword added. The available keywords are as follows:

- @Notes

  Special comments, warnings or information can be highlighted by using this parameter

- @IniFile

  The identifier name of the ini file, that contains the parameter. Or, if no @Key keyword added, the description of the ini file. The default value is 'ParamFile'

- @Section

  The name of the section the contains the parameter Or, if no @Key keyword added, the description of the section. The default value is 'General'

- @Key

  The name of the parameter in the section.

- @Datatype

  Data type of the parameter. The available data types can be found in the ParamDesc.h (TParamDataType)

- @Range

  The validity range of the parameter, it can be finite, infinite, opened, closed interval or enumeration.

- @DefaultValue

  The default value of the parameter

- @VariableName

  If the parameter is not a certain C++ type like int, float, char, etc. then the NAME of the variable should be given by using the special comment.

- @VariableType

  If the parameter is not a certain C++ type like int, float, char, etc. then the TYPE of the variable should be given by using the special comment.

- @Size

  The maximum size of the parameter object. It only works with TAncestorMatrix types. It means that the width and height value should be defined. In case of TString the height should be 1.

- @DependsOn

  The parameter generator has the ability to handle dependencies between the parameters. It means an initialization order can be defined between the parameters. With this option, all of the parameters that have to be initialized before the current one, can be enumerated. They should be given by the full path name. In this case the full path means that the @IniFile, the @Section and the @Key value have to be given in "@IniFile/@Section/@Key" format.

- @Flags

  Flags are the modifiers for the parameter behavior, functionality or visibility.

  - Indirect

    This is a Pointer parameter

  - Invisible

    The parameter is not visible in the parameter grid control

  - Internal

    The parameter data should be stored in the internal memory, it only matters in case of TAncestorMatrix based classes.

  - Constant

    The parameter value can not be changed during the execution.

  - Bii

    The parameter works only on Bi-i systems

  - Interface parameter ID

    This is the name of the constant that specifies this parameter as an interface parameter. The interface parameters are non-visible in the parameter grid control, but they can used to interlace the parameter with a control on the GUI. To access the parameter description structure, the OnSetInterfaceParam event will be triggered during the XML description loading.

4.7.2. Parameter Handling Class

This is the class that handles the parameters on the device side. It adds a secondary buffer for the parameters to prevent the parameter overwriting during the algorithm execution, and also determine an interface to easily initialize and refresh the parameters.

The class contains a pointer of the original parameters of the utilizing class and a buffer for each parameters. The user interface application changes the buffered values and the device side application should call the refresh function of the parameter handling class to override the new parameter value.

If there is no GUI for the application, it uses the TDataFile class to get and set the parameters form the ini files and no double buffering applied.

Note

The enumeration values are stored with different way in **GUI** and **no GUI mode**. In **no GUI mode** they are stored by **value**, in **GUI mode** they are stored by **name**. Therefore, in the ini files which were generated in **GUI mode** the enumeration parameters' name should be replaced by the theirs values to make them usable in **no GUI mode**, on the other direction, the values should be relaced by the names.

4.7.2.1. Functions

The following functions are available in the parameter handling class:

- Constructor

  Allocates the secondary buffers for the parameters and resizes the TAncestorMatrix type parameters.

  In **no GUI mode** it does not allocates secondary buffers.

- Destructor

  Releases the allocated secondary buffers.

  In **no GUI mode** there are no secondary buffers allocated.

- Initialize

  This function initializes the parameter handling. It works only in synchronous host communication mode. So, the function set the host communication mode to synchronous and set back to the original if it finished the execution.

  Send the buffer parameters' addresses to the host, set the pointers of the original parameters and download the initial parameter values from the host.

  In **no GUI mode** there are no secondary buffers allocated. It open the ini files and loads the parameter values directly to the application classes member variables.

- Refresh

  This function refreshes the original parameters with the buffered parameters. It handles the synchronization problem with the host. The parameters will not be mixed during this operation.

  In **no GUI mode** it does the same as the initialize function.

- Update

  This function updates the buffered parameters with the current value of the parameters. It handles the synchronization problem with the host. The parameters will not be mixed during this operation.

  In **no GUI mode** it saves the actual parameter values to the ini files.

- UploadRequest

  This function sends a request to the Host to read the modified parameters from the IV application to it's buffer.

Note

This function overwrites the buffered data. Any non-refreshed data will be lost.

In **no GUI mode** this function does not exist.

4.7.3. Example

The following sample code shows some of the parameter comment blocks. For more details, see the Param sample in the DotNetComponent Samples folder.

Example 4.4. Header Comment Sample, Enumeration Types

```
/**
 @Section Section_Enum
 This the section contains the enumeration types.
*/

/** This is the first enumerator type. This is an implicit number based enumerator

 @Section  Section_Enum
 @Key   Enum_Implicit_Num
 @Range   [0,1,2,3]
 @DefaultValue 2
*/
int Enum1;

/** This is the second enumerator type. This is an implicit character based enumerator
with default values

 @Section  Section_Enum
 @Key   Enum_Implicit_Char
 @Range   [Off,On]
 @DefaultValue Off
*/
int Enum2;

/** This is the fourth enumerator type. This is an Explicit character based enumerator
with custom value

 @Section  Section_Enum
 @Key   Enum_Explicit_Char
 @Datatype DATA_TYPE_ENUM
 @Range   [EnumVal1 = 567 , EnumVal2=123,EnumVal3 =324,EnumVal4= 42342, EnumVal5=35 ]
 @DefaultValue EnumVal3
*/
int Enum4;
```

Example 4.5. Header Comment Sample, Different range types

```
/**
 @Section Section_Range
 This the section contains the range samples.
*/
/**
 This is the first range sample parameter. Two side closed range.
 @Section  Section_Range
 @Key   Range_Close_Close
 @Range   [-2,20]
 @DefaultValue 5
*/
int  Range1;
/**
 This is the fourth range sample parameter. Two size opened range.
 @Section  Section_Range
 @Key   Range_Open_Open
 @Range   (-2,20)
 @DefaultValue 5
*/
int  Range4;
/**
 This is the fifth range sample parameter. Left Infinite, Right Closed.
 @Section  Section_Range
 @Key   Range_Infinite_Close
 @Range   (...,20]
 @DefaultValue 5
*/
int  Range5;
/**
 This is the sixth range sample parameter. Left Closed, Right Infinite.
 @Section  Section_Range
 @Key   Range_Close_Infinite
 @Range   [2,...]
 @DefaultValue 5
*/
int  Range6;
/**
 This is the seventh range sample parameter. Two side infinite.
 @Section  Section_Range
 @Key   Range_Infinite_Infinite
 @Range   [...,...)
 @DefaultValue 5
*/
int  Range7;
```

Example 4.6. Header Comment Sample, Explicit data type parameters

```
/** This is an explicit byte variable.

 @IniFile ExternalParams
 @Key  ExByte
 @Datatype DATA_TYPE_BYTE
 @Range  [0,200]
 @DefaultValue 150
*/
unsigned char  ExByte;
/** This is an explicit unsigned int variable.

 @IniFile ExternalParams
 @Key  ExUInt
 @Datatype DATA_TYPE_UINT
 @Range  [0,3000000000]
 @DefaultValue 2800000000
 @Flags Constant
*/
int  ExUInt;
/** This is an explicit float variable.

 @IniFile ExternalParams
 @Key  ExFloat
 @Datatype DATA_TYPE_FLOAT
 @Range  [-456.456,194.194]
 @DefaultValue 153.505
*/
float  ExFloat;
```

Example 4.7. Header Comment Sample, Matrix parameters

```
/** This is an explicit bit matrix variable. Shows the usage of the dependency
between the parameters.

 @IniFile ExternalParams
 @Section Matrices
 @Key  ExBitMat
 @Datatype DATA_TYPE_BIT_MATRIX
 @Range  [0,1]
 @Size  [5,3]
 @DefaultValue 5 3 0 0 1 0 0 1 1 1 1 1 0 1 0 1 0
 @DependsOn ExternalParams/Matrices/ExFloatMat, ExternalParams/Matrices/ExShortMat
*/
InstantVision::TBitMatrix ExBitMat;
/** This is an explicit byte matrix variable.

 @IniFile ExternalParams
 @Section Matrices
 @Key  ExByteMat
 @Datatype DATA_TYPE_BYTE_MATRIX
 @Range  [1,190]
 @Size  [5,3]
 @DefaultValue 5 3   5 86 97 113 155   16 31 82 40 32   55 54 23 43 65
*/
InstantVision::TByteMatrix ExByteMat;
/** This is an explicit float matrix variable.

 @IniFile ExternalParams
 @Section Matrices
 @Key  ExFloatMat
 @Datatype DATA_TYPE_FLOAT_MATRIX
 @Range  [-456.456,5940.6]
 @Size  [5,3]
 @DefaultValue 5 3   0.4 48 4.48 -5.1 14.1   2.41 8 7.82 24.01 2.12   14.2 53.4 3.4 25.2 -2
 @Flags Internal
*/
InstantVision::TFloatMatrix ExFloatMat;

/** This is an explicit string variable.

 @IniFile ExternalParams
 @Section Matrices
 @Key  ExString
 @Datatype DATA_TYPE_TEXT
 @Size  [64,1]
 @DefaultValue Default text
 @Flags Internal
*/
InstantVision::TString ExString;
```

Example 4.8. Header Comment Sample, Interface and Indirect parameters

```
/** This is an interface variable.

 @IniFile ExternalParams
 @Section SpecialVariables
 @Key  InterfaceVar
 @Datatype DATA_TYPE_INT
 @Range  [0,20]
 @DefaultValue 2
 @Flags  INTERFACE_SAMPLE_VARIABLE
*/
int  InterfaceVariable;

/** This is an interface variable.

 @IniFile ExternalParams
 @Section SpecialVariables
 @Key  IndirectVar
 @Datatype DATA_TYPE_INT
 @Range  [0,20]
 @DefaultValue 2
 @Flags  Indirect
*/
int*  IndirectVariable;
```

5. The Bi-i SDK

5.1. Getting started

The Bi-i SDK is a set of C++ libraries to be used for Bi-i programming. Some parts of the Bi-i SDK are based on classes defined in the BaseData module of the InstantVision libraries. It is assumed that you are familiar with BaseData. See the InstantVision User's Guide and the InstantVision Reference for more details.

5.1.1. The Software Environment

The Bi-i SDK is designed to be used together with Code Composer Studio from Texas Instruments (http://www.ti.com). The required version is 2.2 or above. It is supposed in this manual that you are familiar with Code Composer Studio and C++ programming.

It is recommended to set the "lib" and the "include" folders within the installation folder of the Bi-i software to the C6X_C_DIR environment variable. Normally this is done during the software setup. After that you can use the libraries without path, and the include files with relative path to the include folder.

5.1.2. Running an Application

To run an application on Bi-i first a proper device must be configured with IVConfig, and after that a host application is needed that refers to the device number as it was set in IVConfig. IVRun (IVRun_x86.exe) is a host application that comes with the Bi-i software package, custom host applications can be created using Bi-i API. The IVConfig is described in detail in the Bi-i Manual and the IVRun is described in the InstantVision User's Guide.

When configuring a device to run an application, observe the following guidelines:

- The 'Net' and the 'Authentication' fields must be properly set so that you can connect and log in to the Bi-i.
- The 'Default Application' field must point to the application to be executed. Custom applications based on the Bi-i API and the IVRun may override this default.
- Keep the 'Use AMC/ABC', the 'Debug with JTAG' and the 'Load program from Flash' boxes unchecked.
- Keep the 'Log file' field blank, unless you really want to log Bi-i behaviour at the driver level. Logs from your application will be saved elsewhere.

5.1.3. Memory Usage

Bi-i v2 and v301 is built with a TMS320C6415 DSP having 1 MB of internal memory, which can be used either as data or program memory. The amount of the external memory is 16 MB or 64MB.

On both systems the external memory can be freely used for data storage or program execution, however, the program may run much slower. In general, keep the time critical code and the belonging data in the internal memory. Keep in mind that copying the data to the internal memory, running some calculation and putting back the result to the external memory can be faster than calculating directly on the data in the external memory. The table below summarizes the memory ranges in the Bi-i system in hexadecimal format.

	Bi-i v2 and v301
Internal memory (L2 Cache is disabled)	0x00000000 - 0x000FFFFF
Internal memory (L2 Cache is enabled)	0x00000000 - 0x000BFFFF
External memory	0x80000000 - 0x80FFFFFF (16MB) or 0x80000000 - 0x83FFFFFF (64MB)

Table 5.1. Memory ranges on the Bi-i systems

Most memory ranges can be used freely, but there are some exceptions:

- The space for interrupt vectors must always be 0x00000000 - 0x000001FF.
- When the program is to be written to the flash, the range 0x00000200 - 0x000007FF must be left free for the boot loader.
- Always leave the range 0x80000000 - 0x8000007F free, this is used for host communication.
- Besides the usual input sections, there must be a section called '.host_request', which is an internal administration space for host requests. This can be placed anywhere, usually it's in the external memory, since host operations are never time critical. The space required is less than 64 kB.

5.1.4. Interrupt Vectors

The interrupt vector table occupies the first 128 32-bit words of the memory. It consists of 16 8-word entries, each entry belongs to one interrupt. The table of interrupt vectors is usually an asm file containing exactly 8 instructions for each entries, which is enough for implementing a branch to some code that handles the interrupt. When programming in C/C++ the first entry drives the code after some initialization to the main function like this:

```
RESET_ISR:
MVKL.S2 _c_int00, B0
MVKH.S2 _c_int00, B0
B.S2 B0
NOP
NOP
NOP
NOP
NOP
```

5.1.5. Build Options

Generally the build options of the project can be set freely, but there are some rules to be followed:

- Set the 'Target version to C64XX (mv6400) for Bi-i v2 and v301.
- Always build little endian code (do not use -me).
- In most cases 'Far Calls & Data' must be set (-ml3).
- Setting 'Use Function Subsections' (-mo) is recommended.
- Link the code with the run-time support library, the Bi-i SDK library and the InstantVision libraries.

	Bi-i v2 or v301
Debug	rts6400.lib
	BiiSdk_ccs3_v2d.lib
	IvBase_ccs3_tx64d.lib, IvSi_ccs3_tx64d.lib, IvSif_ccs3_tx64d.lib
Release	rts6400.lib
	BiiSdk_ccs3_v2.lib
	IvBase_ccs3_tx64.lib, IvSi_ccs3_tx64.lib, IvSif_ccs3_tx64.lib

Table 5.2. Recommended libraries

Using the debug libraries is recommended in the development phase. In the release version range checking is disabled in many places.

5.1.6. Namespaces

Inside the Bi-i SDK all classes, functions and types are declared within the 'Bii' namespace. Because of the tight connection with the InstantVision libraries, the 'InstantVision' namespace is also needed.

5.1.7. Initialization and Closing

Before using any other functions from the Bi-i SDK the Init_Library() function must be called. This initializes all the internal data structures for host communication.

The last Bi-i SDK call in the program must be the InstantVision::IVExit() function, which sends a message to the host application telling that the program execution on Bi-i has been finished.

The 'main' macro in Bii.h can perform the initialization and finalization: it replaces the main function with a sequence of Init_Library(), the original main function and Program_End(). Consequently you need not directly call Init_Library() and Program_End(), if Bii.h is included in the file where the main function is implemented. Defining DIRECT_MAIN_FUNC disables this behaviour.

The empty Bi-i SDK application, which only starts and after that stops immediately, is very easy to implement. It includes Bii.h and contains an empty main function. When running it from your host computer a message is received, because of the implicit call of InstantVision::IVExit(), telling to the host application that the program execution on the Bi-i has been finished. When running it from IVRun the Stop button on the toolbar is activated and the Run button is deactivated for a short time (showing that the application is running), but when the 'Stop' message is received, the original status of the toolbar buttons returns (Run is active, Stop is inactive). Two messages appear in the Logs window of IVRun: START and STOP.

Example 5.1. The empty Bi-i SDK application

```
#include "SDK/Bii/Bii.h"

int main(void)
{
return 0;
}
```

5.1.8. Logging

The Bi-i SDK does not have its own logging feature, it uses the logging functionality of InstantVision instead. See the InstantVision documentation for more details.

The LogMsg() function (from InstantVision) can be called from any place in your code, but keep in mind that the program execution is blocked while logging, which takes a few milliseconds. However some logging can be placed even in time critical code, because setting the log level high can disable log messages and the execution time of the LogMsg function can fall far below a microsecond.

Example 5.2. Bi-i SDK application with logging

```
#include "SDK/Bii/Bii.h"
#include "InstantVision/BaseData/Logger.h"

using namespace InstantVision;

int main(void)
{
LogMsg(IVLOG_TRACE,"Hello World !");
return 0;
}
```

In the default case the messages are received by the Host computer (e.g.: Log window of the IVRun application). The SetLogMsgTarget can redirect the messages to the /var/log/messages file on the Axis communication processor.

5.1.9. Stop and Pause Status

The Stop and the Pause status within the Bi-i SDK are internal status indicators that can be set directly from the host application. In IVRun, the Stop and the Pause buttons set the stop and the pause status in the Bi-i respectively. Within the application running on the Bi-i the GetStopStatus() and the GetPauseStatus() functions can retrieve the current value of the Stop and the Pause status.

The example below shows an application that keeps on running, and stops only if the Stop button is pressed in IVRun. After starting it the START message appears in the Logs window, the STOP message comes only when the Stop button was pressed.

Example 5.3. Checking the stop status

```
#include "SDK/Bii/Bii.h"

using namespace Bii;

int main(void)
{
while (!GetStopStatus())
 {
 //Do something here.
 }
return 0;
}
```

5.1.10. Checking the connection

When using the Bi-i as a stand-alone system, a host computer is not required for the normal operation at all. In this case, flash boot mode must be used. See the Bi-i User's Manual for more details.

In the flash boot mode the Bi-i application starts immediately after it has been loaded from the flash memory of the DSP. A Host application (e.g: IVRun) can connect to the Bi-i device also in this case. The following functions help waiting for the connection: WaitForConnection() and CheckConnection() from the Bii.h.

5.2. General Purpose I/O interface (GPIO)

The Bi-i SDK has a general software interface for different GPIO interfaces in the Bi-i cameras. The functions for the GPIO handling is declared in the Include/SDK/Bii/GPIO.h header file (See in the Bi-i Programming Reference). All property of GPIO connectors can be modified separately and all connectors can be written and read separately, but max 32 connectors in the same time. The specific properties for the certain GPIO interfaces are described in the Hardware section of the Bi-i Users Manual documentation (for example: number of connectors, available directions, etc.).

Example 5.4. Usage of GPIO interface

```
//Set data direction to output for the 0 - 7 GPIO pins
SetGPIOProperty(GPIO_Direction, GPIO_out, 0, 8);

//Writes 01010101 data to the GPIO (0 - 7 pins).
WriteGPIO(0x55, 0, 8);

//Set data direction to input for the 2 - 5 GPIO pins
SetGPIOProperty(GPIO_Direction, GPIO_in, 2, 4);

//Reads the 2 - 5 GPIO pins
//The first least significant 4 bits in the data variable will be the result.
unsigned int data = ReadGPIO(2, 4);
```

5.2.1. GPIO v301 component

The GPIO v301 is an additional components in the Bi-i v301 camera. It has the following GPIO channels:

- 8 bidirectional channels. Channel indexes in the software interface: 0 - 7.
- 4 opto coupled output channels. Channel indexes in the software interface: 8 - 11 (in order OUT_A1, OUT_B1, OUT_A2 and OUT_B2).
- 4 opto coupled input channels. Channel indexes in the software interface: 12 - 15 (in order channel 1, 2, 3 and 4).

The directional property is only variable for the first 8 channels, the others have fixed directional. There is a property for pulling up the outputs to 5V (GPIO_Pullup). This property has effect on the first 8 bidirectional channel. Physically two pullup resistor exist on the hardware. Both resistors pulls up 4 channel output (0-3 and 4-7), therefore 4 or 8 channels can be only set in the same time (0-3, 4-7 or 0-7). The default value for all are 1 (connected).

Example 5.5. Usage of GPIO v301 with GPIO interface

```
//Set data direction to output for the 0 - 7 GPIO pins
SetGPIOProperty(GPIO_Direction, GPIO_out, 0, 8);

//Writes 01010101 data to the GPIO (0 - 7 pins).
WriteGPIO(0x55, 0, 8);
LogMsg(IVLOG_INFO, "Value of the GPIO(0-7) after read back: 0x%x", ReadGPIO(0, 8));

//Deactivates pullup resistors.
SetGPIOProperty(GPIO_Pullup, 0, 0, 8);

//Writes opto coupled outputs
WriteGPIO(0x5, 8, 4);

//Reads opto coupled inputs
//The first least significant 4 bits in the result variable will be the received data.
unsigned int result = ReadGPIO(12, 4);
```

Note

The all channels are accessible on a DB-25 connector. See in the Bi-i Users Manual.

5.3. Host Communication

A Bi-i application can access two Host servers. The first runs on the Axis communication chip in the Bi-i hardware. The Bi-i application can refer to this server with the TAxisTarget class. The second one runs on the Host computer which connects to the Bi-i hardware via network (e.g. IVRun application). The Bi-i application can refer to this server with the TNetTarget class. The TNetTarget class is the default target. These classes are implemented in the SDK/Bii/TBiiTarget.h. The Bi-i application access services of input and output with inherited classes of the TServiceAncestor class. See the I/O and I/O Components sections of the InstantVision documentation for more details.

Services use a default data directory which points to a folder of the local file system if the service request does not include a filename with full path. Generally the default data directory may be set in the Host Application (e.g. IVRun). The I/O Components on the Axis communication processor uses the following default directory "/var/data/". This parameter is included in the /etc/axishost.ini file.

Service handler classes in the InstantVison software module	Server on Host computer	Server on Axis communication chip
TStd	supported, this class can send and receive data directly to/from the host application (e.g. IVRun)	supported for operator<< (TString&), these strings are transferred to the /var/log/messages file.
TFile	supported	supported
TMatlab	supported	not supported
TDataFile	supported	supported
TImgIn and TImgOut	supported	supported
THexFrgb and TDShow	supported	not supported
TVideoIn and TVideoOut	supported	not supported
TImgSeqIn and TImgSeqOut	supported	supported

Table 5.3. Supported services on the Bi-i system

Example 5.6. Host Targets

```
TByteMatrix ByteMatrix(156,362);
//Image output service on Axis MCM chip
TImgOut NetImgOut("sample.bmp",TNetTarget());
//Image output service on the Host computer
TImgOut AxisImgOut("sample.bmp",TAxisTarget());

ByteMatrix = 100;
//Creates an image to the default data directory of the Host computer.
NetImgOut << ByteMatrix;
//Creates an image to the default data directory of the Axis.
AxisImgOut << ByteMatrix;
```

A Bi-i application can communicate with I/O Components via host requests. All host requests are stored in a queue located in the memory of the Bi-i device. The processing of host requests consists of the following operations :

- The Bi-i application puts the host request to the net or the Axis host request queue.
- The Bi-i application sends a signal to the Host side showing that there is a new host request.
- The Host side reads the host request from the queue.
- The request is processed on the Host computer or the Axis communication chip.
- The host application or the Axis communication chip sends a signal to the Bi-i application showing that the first item of the net or axis host request queue was completed.

5.3.1. Types of Communication

The Bi-i SDK supports two types of communication between the Bi-i application and the I/O Components: synchronous communication and asynchronous communication.

5.3.1.1. Synchronous Communication

The synchronous communication is the default communication type. The Bi-i application is suspended, when a service handler class sends a synchronous host request, until the signal is received from the I/O Components signing

that the host request was completed. After that the Bi-i application is resumed. The functions and operators of a service handler class that sends synchronous host request are not returned until the requested data is sent or received. In this communication type the net and axis host request queue can contain only one host request at a time.

Example 5.7. Synchronous host communication

```
//Makes a storage class for the image
TByteMatrix ByteMatrix;
//Image input service on the Host computer
TImgIn ImgIn("sample.bmp");

//Sends the host request to the Host computer
ImgIn >> ByteMatrix;
if(ImgIn.good())
    {
    // The requested data is available in the ByteMatrix object
    }

else
    {
    // The requested data is not available because the host request was failed.
    }
```

5.3.1.2. Asynchronous Communication

The asynchronous communication is an advanced communication type, which enables code execution on the Bi-i while processing net or axis host requests are in progress on the I/O Components. The Bi-i application puts the asynchronous host request to the axis or net host request queue and sends a signal to the I/O Components. The Bi-i application does not wait for the signal from the Host server, meaning that host requests can aggregate in the net and axis host request queue. All service handler classes use the same communication type at a given time for the same I/O Components, because they all share a single net and axis host request queue. Using this communication type is recommended only for advanced users. Keep in mind the following points:

- Do not send host requests in a tight loop, because the net or axis host request queue gets filled or the communication processor in the Bi-i device gets overloaded.
- The data to be sent to the Host server must be available at the time when the host request is processed. This can be significantly later than when the function or operator of a service handler class was called.

The following functions are implemented in the SDK/Bii/Bii.h:

SetAsyncMode()	Enables or disables the asynchronous communication mode for the selected interface (net/axis).
GetAsyncMode()	Returns the current communication mode for the selected interface (net/axis).
GetHostCnt()	Returns the number of pending host requests for the selected interface (net/axis).
WaitHostCnt()	Waits until the number of pending host requests is lower then or equal to a count for the selected interface (net/axis).
GetMaxHostCnt()	Returns the size of the net or axis host request queue.

Example 5.8. Asynchronous host communication

```
int i, loop_cnt;
//Creates matrices
TByteMatrix MatrixArray[10];
//Initializes matrices
for(i=0;i<10;i++) MatrixArray[i].Resize(26,13) = i*20;

//Creates image sequence service interface for the Axis
TImgSeqOut NetSeq("Sample.bmp",TNetTarget());
//Creates image sequence service interface for the Host computer
TImgSeqOut AxisSeq("Sample.bmp",TAxisTarget());

//Enables the asynchronous communication for the Axis
SetAsyncMode(true,TNetTarget());
//Enables the asynchronous communication for the Host computer
SetAsyncMode(true,TAxisTarget());

//Sends twenty host requests.
for(i=0;i<10;i++)
    {
    //Creates a numbered image sequence
    //in the file system of the Host computer.
    NetSeq << MatrixArray[i];
    //Creates a numbered image sequence
    //in the file system of the communication processor.
    AxisSeq << MatrixArray[i];
    }

loop_cnt = 0;
//Waits until the twenty host requests are finished.
while ((GetHostCnt(0,TNetTarget()) > 0) ||
       (GetHostCnt(0,TAxisTarget()) > 0))
    {
    //This code segment runs parallel with the host requests
    loop_cnt++;
    }
```

5.4. TIbis

The Bi-i has an Ibis5 grayscale or color CMOS image sensor with 1280 x 1024 resolution and 40 MHz pixel clock. TIbis is a class in the Bi-i SDK that represents the Ibis5 sensor.

When working with the TIbis class, some elements of the BaseData module in the InstantVision libraries must be used:

- The InstantVision::TByteMatrix class is the representation of an 8-bit gray-scale image in the memory of the DSP. This class is used when reading the image from the sensor into the DSP memory.
- The InstantVision::TRect structure defines a rectangle with the Top, Left, Width and Height elements. This structure can be used when selecting a rectangle on th sensor, from which the image will be read.

See the InstantVision documentation for more details.

5.4.1. Framerate

The time required for reading an image from the sensor can be calculated as:

```
height * (width * 25ns + row blanking time)
```

Height is the number of lines on the image, width is the number of pixels in each line. Row blanking time can be 3.5 us 7 us, 14 us or 28 us, depending on the IBIS_GRAN_X_SEQ parameter. When calculating the maximum available framerate the shutter period must be also added (except in rolling shutter mode, where the shutter and read-out run at the same time).

5.4.2. Acquiring Images the Simple Way

The simplest way images can be acquired from the Ibis5 sensor is just a few program lines:

- Declare a TIbis object and a TByteMatrix object, preferably with the same size as the image rectangle on the sensor.
- Set the image rectangle on the TIbis object with the SetRect() function. Any rectangle can be set within the physical dimensions of the sensor, the only limitation is that the horizontal position and size must be even. Image coordinates are zero based.
- Set the shutter period on the TIbis object with the SetShutter() function.
- Use operator >> on the TIbis object to take an image and to read it into the DSP memory.

Example 5.9. Taking an Image using operator >> on a TIbis object

```
TByteMatrix img(640, 480); // The object for the image, 640x480 pixels
TIbis ibis; //The object for Ibis5

//Setting the image rectangle on Ibis5: 640x480 pixels from position (100,100)
ibis.SetRect(100, 100, 640, 480);

//Setting the shutter period to 5000 us
ibis.SetShutter(5000);

//Taking an image
ibis >> img;
```

5.4.3. Parallel Operations

During the shutter period and during the image read-out other operations can run on the DSP. If you want to do this, call ShutterStart and begin other operations. Later use the ShutterCheck() function to test whether the shutter has already finished. Alternatively you can wait for the end of the shutter period with ShutterWait(). For the read-out period similar functions can be used: ReadStart(), ReadCheck() and ReadWait().

Example 5.10. Parallel operations on the Ibis5 sensor

```
//Starting the shutter
ibis.ShutterStart();

//Running some other code while checking the shutter status periodically
while (!ibis.ShutterCheck())
 {//Do something else here, while the shutter is not yet finished on Ibis5}

//Starting the image read-out
ibis.ReadStart(img);

//Running some other code while checking the read status periodically
while (!ibis.ReadCheck())
 {//Do something else here, while the image is being read from Ibis5}
```

5.4.4. B Mode

Normally the sensor directly controls the shutter period. The length of the shutter is set with SetShutter() and the operation is started with ShutterStart() (or Shutter, or operator >>, they all call ShutterStart() inside). After that, the shutter is finished exactly after the desired period, which is completely independent of the DSP. However there is another operation mode, which is called B mode. The name comes from the similar operation on manual cameras: in B mode the shutter is finished only when the exposure button is released. Similarly for the TIbis object the shutter can be started with ShutterStartB() (like pressing the exposure button) and stopped with ShutterStopB() (like releasing the exposure button). The time between the two functions is controlled in your code only. Setting the shutter period with SetShutter() has no effect in B mode.

The B mode can be useful if extremely long shutter period is needed. The longest possible shutter in normal mode (that can be set with SetShutter) is 26208 us due to the limitation of the sensor. Note that the image can become noisy when using very long shutter. The B mode is available on Bi-i v2 only.

Example 5.11. B mode shutter on the Ibis5 sensor

```
///Starting the shutter
ibis.ShutterStartB();

//Wait here for the end of the shutter period

/Stopping the shutter
ibis.ShutterStopB();
```

5.4.5. Subsampling

When using subsampling, the image read-out works a different way: only the half of the pixels is read. As a result, a half sized image can be taken from the same physical area of the sensor. Subsampling can be set horizontally, vertically or in both directions. Setting the IBIS_X_SUBSAMPLE and the IBIS_Y_SUBSAMPLE parameter to 1 enables the horizontal and the vertical subsampling respectively.

Subsampling works on patterns containing four pixels: two pixels are always read and the remaining two pixels are skipped. The IBIS_X_SWAP12 and IBIS_X_SWAP30 parameters control the horizontal patterns, while the IBIS_Y_SWAP12 and IBIS_Y_SWAP30 parameters set the vertical patterns. The table below shows the horizontal patterns. 'X' means that the pixel is read, 'O' means that the pixel is skipped. Vertical patterns are similar.

IBIS_X_SUBSAMPLE	IBIS_X_SWAP12	IBIS_X_SWAP30	Pattern
0	no effect	no effect	XXXX
1	0	0	XXOO
1	1	0	XOXO
1	0	1	OXOX
1	1	1	OOXX

Table 5.4. Horizontal patterns for Ibis5 subsampling

There are some special rules when setting the image rectangle with subsampling:

- The width and the height means the number of pixels that are read, the number of pixels covered on the sensor is twice as much. Consequently left+2*width and top+2*height must be within the sensor area.
- The left parameter must be a multiple of four.

Example 5.12. Setting subsampling both directions on Ibis5

```
/*Setting subsampling horizontally,
swap12 will result XOXOXOXO pattern*/
ibis.Set(IBIS_X_SUBSAMPLE, 1);
ibis.Set(IBIS_X_SWAP12, 1);

//The same thing vertically
ibis.Set(IBIS_Y_SUBSAMPLE, 1);
ibis.Set(IBIS_Y_SWAP12, 1);
```

5.4.6. Rolling Shutter

In rolling shutter mode there is no separate shutter and read-out phase on the sensor, the shutter and the read-out runs at the same time but on different lines of the image. Consequently, the available framerate in rolling shutter mode is higher than in synchronous shutter mode. On the other hand, different lines of the image are taken at

different times, which can cause side effects. Siginificant distortion can appear when taking images of fast moving objects, or a short flash can be seen only in some lines. Setting the IBIS_ROLLING_SHUTTER parameter to 1 enables the rolling shutter mode.

The shutter period (i.e. the IBIS_INT_TIME parameter) in rolling shutter mode refers to the number of lines on the image between the shutter and the read-out. Thus the minimum available shutter period is the time required for reading a line, and the maximum available shutter period is the time required for reading the complete image except the last line (see the Framerate section). The resolution between the minimum and the maximum is the time for reading one line. The minimum, the maximum and the resolution depends on the size of the image. The proper value is set to IBIS_INT_TIME when calling the SetShutter with the required shutter period in us. It is assumed that IBIS_ROLLING_SHUTTER, the size of the image and IBIS_GRAN_X_SEQ are already set when SetShutter is called.

In rolling shutter mode the images must be read periodically without significant delay, because the delay between the images increases the actual shutter period for the first few lines on the image. It means that the framerate is fixed, and it depends on the size of the image. Another similar effect is that the first few lines of the first image are blank because there was no shutter for those lines at all. The number of lines affected depends on the IBIS_INT_TIME parameter in both cases. A simple, but useless code for taking images in rolling shutter mode looks like this:

Example 5.13. Rolling shutter mode on Ibis5 - USELESS CODE

```
//Setting to rolling shutter mode
ibis.Set(IBIS_ROLLING_SHUTTER, 1);

//Setting the image rectangle
ibis.SetRect(IMG_LEFT, IMG_TOP, IMG_WIDTH, IMG_HEIGHT);

//Setting the shutter period
ibis.SetShutter(SHUTTER_US);

//Reading hte images in a loop
//THEORETICALLY IT WORKS BUT PRACTICALLY IT IS USELESS
while (some_condition)
 {
 ibis.Read(img);
 }
```

The problem with this implementation is that there is now place to write any code that can process the image. If we simply write it after the Read() function, then the time for the processing increases the shutter period for a part of the next image. In a usable code two image buffer are nedded: one is always filled from the sensor, and the other can be processed. Instead of the Read() function use ReadStart() at the beginning of the loop, and ReadWait() at the end.

Example 5.14. Rolling shutter mode on Ibis5

```
//Images are read here from Ibis5
TByteMatrix img1(IMG_WIDTH, IMG_HEIGHT), img2(IMG_WIDTH, IMG_HEIGHT);
//Points to the buffer where the current image is read from Ibis5
TByteMatrix *readptr;
//Points to the buffer that contains a ready image to be processed
TByteMatrix *procptr;
//Only used when changing readptr and sendptr
TByteMatrix *tmpptr;
//The object for Ibis5
TIbis ibis;

//Setting to rolling shutter mode
ibis.Set(IBIS_ROLLING_SHUTTER, 1);

//Setting the image rectangle
ibis.SetRect(IMG_LEFT, IMG_TOP, IMG_WIDTH, IMG_HEIGHT);

//Setting the shutter period
ibis.SetShutter(SHUTTER_US);

//Two images are read. The first image is never perfect in rolling shutter mode.
ibis.Read(img1);
ibis.Read(img1);

//Initializing the pointers, now we can read to img2 and we can process img1
readptr = &img2;
procptr = &img1;

//The loop
while (some_condition)
 {
 //Starting read-out to readptr
 ibis.ReadStart(*readptr);

 //Process the image at procptr here

 //Exchanging readptr and procptr
 tmpptr = readptr;
 readptr = procptr;
 procptr = tmpptr;

 /Waiting for the end of the current image read-out
 ibis.ReadWait();
 }
```

5.4.7. Subwindows

The subwindows is a multi-window readout method from a main previously set rectangle on the Ibis5. This works only with Bi-i v2 Stereo HRES and the Bi-i v301. This is a two step readout, because a row can be read only once from the Ibis5 and the subwindows can overlap. Therefore the rows are read to a row buffer, and in the second step the data will be copied to the corresponding rows of the subwindows.

Important:

- The subwindows must be inside the main rectangle.
- The minimum width of the subwindows is 200, if width is set to a lower value then the algorithm will read more pixels.
- The positions of the subwindows in x direction must be dividable by 4.

Before reading the image an initialization step is needed. This calculates the effective rectangles of the readout and stores these in a descriptor structure. Therefore, if the size and position of the subwindows on the rectangle are not changed, the initialization step is needed only once.

Note

When using many subwindows, the time of the readout of a row may be longer than the available time (before the readout of the next row is started). In this case an assert will occur. Increasing the row blanking time (Set(IBIS_GRAN_X_SEQ, x)) can solve this problem.

Example 5.15. Subwindows readout on Ibis5

```
//Number of subwindows
#define SWNUM 6

//Parameters of main rectangle
#define MAIN_WIDTH 800
#define MAIN_HEIGHT 600
#define MAIN_LEFT 0
#define MAIN_TOP 0

//Memory space for descriptors.
//The CalcDescArraySize macro calculates the needed size of array.
unsigned char DescArray[CalcDescArraySize(SWNUM)];
//Memory space for row buffer.
unsigned char RowBuffer[MAIN_WIDTH];

TRect mainRect = {MAIN_LEFT, MAIN_TOP, MAIN_WIDTH, MAIN_HEIGHT};
TByteMatrix subwindow[SWNUM];
TPair_Int position[SWNUM];
TIbisDual ibis;
unsigned int i;

//Sets initial parameters of subwindows.
for (i = 0; i < SWNUM; i++)
{
subwindow[i].Resize(200, 200);
position[i].x = 80*i;
position[i].y = 80*i;
}

//Sets shutter time and main rectangle
ibis.SetShutter(5000);
ibis.SetRect(mainRect);

//Gets sub images from Ibis.
ibis.Shutter();
ibis.Subwindows(subwindow, position, SWNUM, DescArray, RowBuffer);
//OR
ibis.SubwndInitialize(subwindow, position, SWNUM, DescArray);
ibis.Shutter();
ibis.SubwndReadout(RowBuffer);
```

Note

If DescArray and RowBuffer are not specified, then these will be allocated automatically. In this case, or if the specified arrays are in the external memory, the running time will be longer.

5.4.8. Auto Shutter

There is a feature in the TIbis, which calculates and sets on automatically shutter time for the actual rectangle, in a way that the average of the image is equal to the desired value.

Note

If the lighting or the scenery change, the average of the acquired image changes as well.

Example 5.16. Auto Shutter

```
TByteMatrix img(1280, 1024); // The object for the image
TIbis ibis; //The object for Ibis5
unsigned int ShutterTime; //It will be set by AutoShutter

//Setting the shutter time that the average of images should be 120.
ShutterTime = ibis.AutoShutter(120);

//Taking an image
ibis >> img;
```

5.4.9. Using the Stereo HRES

Only Bi-i v2 has Stereo HRES version. In this configuration the Bi-i is equipped with two Ibis5 sensors that are capable to aquire images simultaneously.

Important note:

- The shutter time must be the same in both sensors. If the previously set shutter times are different, then the images will be acquired using the shorter shutter time.
- The image readout from the sensors is not allowed in the same time.
- The TIbis class does not work on this hardware. There is a derived class from the TIbis class, called TIbisDual, which inherits all feature of the TIbis class.

There is a function for sensor selection (Select()). Left, right or both sensors can be selected. The constructor set both sensors to default and after this, the left sensor will be selected.

Example 5.17. Acquiring Images from the Stereo HRES

```
//Image width from left Ibis.
#define IMG_WIDTH1 100
//Image height from left Ibis.
#define IMG_HEIGHT1 100
//Image width from right Ibis.
#define IMG_WIDTH2 640
//Image height from right Ibis.
#define IMG_HEIGHT2 480
//Integration time for both Ibis.
#define SHUTTER_US 1000

//The buffers where the images are read: same size as the rectangle on Ibis5
TByteMatrix img1(IMG_WIDTH1, IMG_HEIGHT1);
TByteMatrix img2(IMG_WIDTH2, IMG_HEIGHT2);
//The object for Ibis5
TIbisDual ibis;

//Set image size on left Ibis
ibis.Set(IBIS_NROF_PIXELS, IMG_WIDTH1);
ibis.Set(IBIS_NROF_LINES, IMG_HEIGHT1);
//Select right Ibis
ibis.Select(IBIS_SELECT_RIGHT);
//Set image size
ibis.Set(IBIS_NROF_PIXELS, IMG_WIDTH2);
ibis.Set(IBIS_NROF_LINES, IMG_HEIGHT2);

//Set shutter speed and shutter with both Ibis in same time
ibis.Select(IBIS_SELECT_BOTH);
ibis.SetShutter(SHUTTER_US);
ibis.Shutter();
//Read image from left Ibis
ibis.Select(IBIS_SELECT_LEFT);
ibis.Read(img1);
//Read image from right Ibis
ibis.Select(IBIS_SELECT_RIGHT);
ibis.Read(img2);
```

5.4.10. Flash control

There is a connector in the GPIO v301 interface as a flash control output signal which controlled by TIbis class. The flashing is turned off in default which means that this signal is holded in high (5V). If the flash is turned on the signal goes low at shutter start and goes back high at the end of the shutter. In the SDK you can only turns on/off the flash and gets the flash status. The flash signal is controlled by hardware.

Note

It works only on Bi-i v301 with GPIO v301 interface. The flash output is the specific connector in the GPIO v301 interface (See in the Bi-i Users Manual / Hardware section).

Example 5.18. Setting flash

```
//Creates TIbis object
TIbis Ibis;

//Turns the flash on
Ibis.SetFlash(true);

//Check the flash status
LogMsg(IVLOG_INFO, "Actual flash status: %d", (int)Ibis.GetFlash());
```

5.4.11. Triggered shutter

The shutter procedure can be started from an external trigger signal. This is the triggered shutter which is same as the conventional synchronous shutter but after shutter start (in SDK code) the Ibis waits for a trigger signal (falling

edge) and start the effective shutter after it. The trigger input is the specific connector in the GPIO v301 interface (See in the Bi-i Users Manual / Hardware section).

The waiting for trigger can be canceled. In this case, the whole shutter procedure is also canceled.

Note

It works only on Bi-i v301.

Example 5.19. Triggered shutter

```
//Creates TIbis object.
TIbis Ibis;

/////// Normal shutter with triggering.

  //Aquiring image with triggered shutter.
  Ibis.TriggeredShutter();

/////// Shutter with triggering but if a condition is realized then the shutter is canceled.

  //Starts aquiring
  Ibis.TriggeredShutterStart();

  //Cancels the shutter.
  if(Condition)
     Ibis.TriggeredShutterCancel();

  //Wait for the end of the shutter.
  Ibis.ShutterWait();
```

5.5. TACE and TACE_IPL

Bi-i v2 has optionally an ACE16k (Focal Plain Array Analog Processor) chip. ACE16k contains a low resolution (128*128) CMOS grayscale image sensor and an analog processor array. (One processor belongs to one pixel of sensor without A/D conversion.) All processor unit are connected to the neighbouring processors with specified weight. These weights represent a general analog program (template). This processor array is more useful for several image processing operations, than the traditional processors, because it processes the whole image in parallel.

The TACE class contains all basic functionality for control of ACE16k chip (like image and template transfer, logic, template execution, etc.). The constructor initializes the program memory of ACE16k, and write some template to the end of template memory for basic operations. Therefore these template memories (last 8) are reserved.

The TACE_IPL class is an image processing library for ACE16k .

Note

Some functionalities of the BaseData module in InstantVision (TByteMatrix, TBitMatrix) are required for using of TACE and TACE_IPL.

5.5.1. I/O Interface

The ACE16k has a 32-bit wide digital interface, which is able to transfer images and templates between ACE16k and the DSP.

The internal memory of ACE16k consists of

- 8 grayscale image memories (TClam enumeration: C_LAM1 - C_LAM8),

- 2 binary image memories (TCllm enumeration: C_LLM1 - C_LLM2),
- 32 template memories (TCtem enumeration: C_TEM1 - C_TEM32).

Two kind of images can be transferred: binary images from/to a TBitMatrix in the DSP memory, and grayscale images from/to a TByteMatrix in the DSP memory. Images can be read or written with image transfer functions or with operators. There are separate transfer functions for binary images, gray scale images and templates. However, the << and >> operators can be used for all data types, the actual operation depends on the type of the right side operand.

Member functions for image and template transfer

- TACE::WriteLAM: Writes a grayscale image to ACE16k from an InstantVision::TByteMatrix on the DSP memory.
- TACE::WriteLLM: Writes a binary image to ACE16k from an InstantVision::TBitMatrix on the DSP memory.
- TACE::WriteTEM: Writes a template to ACE16k from a Bii::TAce16kTem on the DSP memory.
- TACE::ReadLAM: Reads a grayscale image from ACE16k to an InstantVision::TByteMatrix on the DSP memory.
- TACE::ReadLLM: Reads a binary image from ACE16k to an InstantVision::TBitMatrix on the DSP memory.
- TACE::ReadTEM: Reads a template from ACE16k to a Bii::TAce16kTem on the DSP memory.

Note

Not only the whole image can be read from ACE16k. The readout image size depends on the size of the target TByteMatrix or TBitMatrix. The number of column must be 128, but the number of rows can be lower than 128. In addition, the starting row can be specified in a separate parameter. In contrast, only images of 128x128 pixels can be written to ACE16k, completely filling a C_LAM or C_LLM.

Operators for transfers:

```
Write operator: (TACE object) << (TByteMatrix, TBitMatrix or TAce16kTem);
Read operator: (TACE object) >> (TByteMatrix, TBitMatrix or TAce16kTem);
```

When calling the Read or Write functions, the source or target memory in ACE16k is specified in a separate parameter. In contrast, the source or target ACE16k memory of the >> and << operators must be set prior to calling the operator. Use the RdID() function to set the the source memory for reading and the WrID() function to set the target memory for writing. ID() function sets both the source and the target. Calling the << operator with C_LAM, C_LLM or C_TEM on the right side is equivalent to calling the ID() function.

Note

These operators can be chained, because the return value of all operators is the reference of the TACE object .

Example 5.20. Setting the current IDs on ACE16k

```
TACE ace;

//Setting of current gray scale image ID for reading.
ace.RdID(C_LAM3);
//Setting of current binary image ID for writing.
ace.WrID(C_LAM2);
//Setting of current binary image ID for reading and writing.
ace.ID(C_LLM2);
//or
ace << C_LLM2;
```

This operator can be chained:

```
//Setting of gray scale and binary image ID and template ID for reading and writing.
ace << C_LAM3 << C_TEM 2 << C_LLM2;
```

The example below demonstrates several image transfer operations.

Example 5.21. Image transfer between the DSP and ACE16k with TACE

```
TACE ace;
TByteMatrix GrayImage(128,128);
TBitMatrix BinImage(128,128);

//Writing gray scale image to C_LAM2
ace.WriteLAM(C_LAM2, GrayImage);
//or
ace << C_LAM2 << Image;

//Reading gray scale image from C_LAM2
ace.ReadLAM(GrayImage, C_LAM2);
//or
ace << C_LAM2 >> GrayImage;

//Reading binary image from C_LLM1;
ace.ReadLLM(BinImage, C_LLM1);
//or (C_LLM1 as default target and source is set by the constructor)
ace >> BinImage;

//Reading a gray scale 128*50 subimage from C_LAM1 started from 20th row
TByteMatrix SubImage(128, 50);
ace.ReadLAM(SubImage, C_LAM1, 20);
```

5.5.1.1. Template Structure and Template Transfer

The TAce16KTem structure represents a template in the DSP memory. The TAce16KTem structure is declared in the TAce16kTem.h header file. The corresponding template structure in the file system of the host computer is a text file containing one or more template structures in the following format:

```
[Template_1]

Control = 9 1 -0.75 0 0.75 -1.5 0 1.5 -0.75 0 0.75
Feedback = 9 1 0 0 0 0 -6.0 0 0 0 0
Current = 3.0
Bias = 0
Mode = 3
References = 11 1 128 0 178 229 128 142 255 112 128 37 23
```

The number between the square brackets is the identifier of the template data. A template contains the following data:

Control — 3x3 control matrix. The first two values sign that this field contains 9x1 elements. The range of these elements (except the middle element) is -3.0 - 3.0. The middle element's range is -6.0 - 6.0.

FeedBack	3x3 feedback matrix. The first two values sign that this field contains 9x1 elements. The range of these elements except the middle element is -3.0 - 3.0. The middle element's range is -6.0 - 6.0.
Current	Spatial invariant offset value. the range is -6.0 - 6.0.
Bias	Weight for spatial variant offset image. the range is -6.0 - 6.0.
Mode	Template execution mode. This can be 1, 2, 3 or 4.
References	11 analog references for template (optional). The first two values sign that this field contains 11x1 elements. If not specified, after loading in the DSP the first reference value will be -1. If this template is written to ACE16k, references will be a set of recommended references in the template memory of ACE16k. The order of these is same as in the TAce16kTem structure.

Note

Recommended reference values are: 128, 0, 178, 229, 128, 167, 140, 109, 128, 47, 23.

The last 8 template memories in the ACE16k are reserved for some basic operations, thus do not overwrite these memories. Many ACE16k operations (including image transfer) depend on these templates.

Example 5.22. Template transfer with TACE

```
TACE ace;
TAce16kTem Template;

//Writting template to ACE16k
ace.WriteTEM(C_TEM1, Template);
//or
ace << Template;

//Reading template from ACE16k
ace.ReadTEM(Template, C_TEM1);
//or
ace >> Template;
```

Note

Before writing the template to ACE16k, the template must be loaded from the host computer to the DSP or generated on the DSP.

5.5.1.2. Loading Templates from the Host Computer

The template file is a special ini file (see above), which has fix variables. There is a class which sends and receives templates to or from a template file. This class is inherited from the TDataFile class.

Example 5.23. Loading templates from the host computer

```
#include "SDK/Bii/Ace16k/TAceTemplate.h"
#include "SDK/Bii/Ace16k/TAce16kTem.h"

//Creating an object for loading a template from sobel.tem ini file
TAceTemplate TemFile("sobel.tem", DF_CREATE);

//Declaring a structure for template in DSP memory
TAce16kTem Template;

//Loading the template from "Template_1" section in the ini file
TemFile.GetTem("Template_1", Template);
```

5.5.2. Internal Image Transfer

Images can be transferred between two internal image memories of the ACE16k. Both the source and the target can be binary or gray scale image. If both image memories in transfer operation is same type, this operation is called image copy. If the type of image memories are different, it is called image conversion.

Example 5.24. Internal image copying with TACE

```
TACE ace;

//Grayscale image copying from C_LAM1 to C_LAM2
ace.CopyLAM(C_LAM2, C_LAM1);

//Binary image copying from C_LLM1 to C_LLM2
ace.CopyLLM(C_LLM2, C_LLM1);
```

There is a special binary image copying called exchange LLMs. This member function has no parameters, because there are only two binary images are on the ACE16k. This operation exchanges the contents of the two binary image memories.

```
//Exchanging binary image memories
ace.ExchangeLLM();
```

With conversion methods of TACE, binary images can be converted to gray scale image and vice versa. When converting gray scale to binary, the pixel values above a given threshold are converted to 1, others are converted to 0. When converting binary to gray scale, 0 is converted to 0, while 1 is converted to 255.

Example 5.25. Image conversion with TACE

```
TACE ace;

//Converting from binary to gray scale
ace.ConvLLMtoLAM(C_LAM1, C_LLM1);

//COnverting from gray scale to binary with 128 threshold value
ace.ConvLAMtoLLM(C_LLM1, C_LAM1, 128);
```

5.5.3. Image Initialization

Both gray scale and binary images can be filled (initialized) with a specified value using the Set functions. In addition the << operator with an integer fills the current C_LAM with a constant value, while calling the << operator with a logical value fills the current C_LLM.

Example 5.26. Initialization of image memories with TACE

```
TACE ace;

//Initializing a gray scale image memory with 100 gray scale pixel value.
ace.SetLAM(C_LAM2, 100);
//or
ace << C_LAM2 << 100;

//Initializing a binary image memory with false binary value.
ace.SetLLM(C_LLM2, false);
//or
ace << C_LLM2 << false;
```

5.5.4. Operations on Images

The ACE16k can processes images with several operations like logic on binary images, template execution on binary or gray scale images, etc. The IPL is based also on these operations.

5.5.4.1. Bit-Wise Logic on Binary Images

The logical unit of the processor array is the only digital module in the ACE16k. This performs a bit-wise logical operation on binary images, but the output image can be also an analog gray scale image, therefore the output parameter is a reference of an object of a helper class, called TCImageOut which is a container for a specified image memory ID. For all logic operations except 'not' the input images are always C_LLM1 and C_LLM2, therefore these operations have no input parameters.

Example 5.27. Bit-wise logic operations with TACE

```
TACE ace;

//Logical XOR operation to a binary output image memory
ace.Logic(C_LLM2, ACE_LOG_XOR);

//Logical XOR operation to a gray scale output image memory
ace.Logic(C_LAM5, ACE_LOG_XOR);

//Not Operation from C_LLM1 to C_LLM2.
ace.Not(C_LLM2, C_LLM1);
```

5.5.4.2. Template Execution and TACEtransient class

The template execution (also known as transient evolution) is an analog operation in ACE16k. Te program for this operation is the template. This program contains the weights for connecting the processor units and the internal analog voltage references. The result is depends on the running time of the operation.

There are four basic modes of template execution:

Template execution modes (Mode parameter in the template structure)

1. Template with feed back and without spatial variant offset. All elements of feedback matrix have effect: the output values of neighboring processors are fed back to the input of middle processor unit with the specified weights. In the control matrix, only the middle element has effect: the input values of the neighboring processors are not connected to the input of middle processor unit.

2. Template with feed back and with spatial variant offset. This is same as the mode 1, however it has a spatial variant offset with the weight of the bias element of the template.

3. Template with feed forward and without spatial variant offset. In the feedback matrix, only the middle element has effect: only the own output is fed back to the input. All elements of control matrix have effect: the input values of neighboring processors are connected to the input of middle processor unit.

4. Template with feed forward and with spatial variant offset. This is same as the mode 3, however it has a spatial variant offset with the weight of the bias element of the template.

The control of template execution is performed by a helper class called TACEtransient. This class contains all parameters that are needed for this operation, and stores the pointer to the corresponding TACE object. The Set() member function sets base parameters of execution, but there are some parameters, which can be changed separately (These are set by Set function to default). These functionality are used for more complex operations.

- Boundary condition: Working mode of cells (processor units) on the bound of processor array.
- Writing mask: Output analog memory writing mask.
- Freezing mask: Evolution enabling mask.
- Iteration: Number of transients in the operation.

The example below shows a continuous time template execution on a gray scale image with writing mask. It runs for 2 microseconds and stops. This example is useful for gray scale operations such as sobel filter laplace, inversion, etc.

Example 5.28. Continuous time template execution with TACEtransient

```
TACE ace;
TACEtransient Transient(&ace);
TByteMatrix GrayImage(128, 128);
TBitMatrix BinImage(128, 128);
TAce16kTem Template;

//Loading images and template

//Writing template to ACE16k
ace << C_TEM1 << Template;
//Writing input gray scale image to ACE16k
ace << C_LAM1 << GrayImage;
//Writing mask image to ACE16k
ace << C_LLM1 << BinImage;

//Setting parameters of transient
Transient.Set(C_LAM1, C_TEM1, C_LAM1, C_LAM1, C_NONE, false);
Transient.WritingMask.Set(C_LLM1);

//Running transient evolution
Transient.Run(2.0);
```

The example below shows an iterative template execution on a binary input image. It runs for 1 microseconds in each iteration cycles. After each iterations the output is copied back to the input and to the state. The number of iterations is 10. This example is a typical morphological operation.

Example 5.29. Iterative template execution with TACEtransient

```
TACE ace;
TACEtransient Transient(&ace);
TByteMatrix GrayImage(128, 128);
TBitMatrix BinImage(128, 128);
TAce16kTem Template;

//Loading images and template

//Writing template to ACE16k
ace << C_TEM1 << Template;
//Writing input image to ACE16k
ace << C_LLM1 << BinImage;

//Setting parameters of transient
Transient.Set(C_LAM1, C_TEM1, C_LLM1, C_LLM1, C_NONE, true);
Transient.Iteration = 10;

//Running transient evolution
Transient.Run(1.0);
```

5.5.4.3. Diffusion

There is a resistive grid among the elements of processor array. When activating it, the voltage levels representing the value in analog memory cells, are equalized between the neighboring pixels. In other words the resistive grid performs a diffusion operation on the image: the rate of the diffusion depends on the duration of the operation. Because of the architecture, the diffusion is extremely fast.

Example 5.30. Diffusion with TACE

```
TACE ace;

//Diffusing C_LAM1 to C_LAM2 to 500 nanoseconds.
ace.Diffusion(C_LAM2, C_LAM1, 0.5);
```

5.5.5. Capturing the Optical Input

The optical sensor uses the photo generated current on an N-well/P-Substrate junction to discharge a capacitor that has been previously precharged. The precharge value is a parameter of capture function. The recommended value for precharging is 195. If it is changed the contrast of captured image will be changed.

The output image of the capturing from image sensor can be stored in a binary or gray scale image memory. When the target is a binary memory, the captured image will be thresholded with the 128 gray pixel value. There are two member data for capturing in the TACE class. There are set and get functions for them:

- Integration time (The constructor of TACE sets it to 500 microseconds)
- Precharge value (The constructor of TACE sets it to the recommended value: 195)

Example 5.31. Image capturing on ACE16k with TACE

```
TACE ace;

//Setting of capturing parameters
ace.Set_Int_Time(1000); //Integration time = 1 millisecond
ace.Set_Precharge(195); //Precharge value  = 195

//Capturing to C_LAM1
ace.CaptureOpt(C_LAM1);

//Capturing to C_LLM1
ace.CaptureOpt(C_LLM1);
```

5.5.6. Image Processing Library (IPL)

Image processing library for ACE16k is implemented in a class called TACE_IPL, which contains two function groups for processing images: morphological operations and gray scale operations. The TACE_IPL is derived from TACE class and inherits all the functionality. The constructor of this class initializes the needed instruction group, writes corresponding IPL templates to the ACE16k. Therefore the C_TEM1-C_TEM21 templates are reserved TACE_IPL. Overwriting these templates corrupts the TACE_IPL functionality. The IPL mode can be changed after creating a TACE_IPL object with the SetIPLMode() member function.

Note

If the morphological group is used a calibration step is needed for the correct working in about every 10 milliseconds.

Example 5.32. Typical morphological operation with TACE_IPL

```
TACE_IPL ace(IPL_MORPH);
TBitMatrix BinImage(128, 128);

//Loading input image

//Writing image to ACE16k
ace << C_LLM2 << BinImage1;

//Calibration for morphology
ace.Calibrate();

//Edge detection with full connectivity (BOX_MASK)
ace.Edge8(1);

//Reading result images
ace >> BinImage2;
```

Example 5.33. Typical gray scale operation with TACE_IPL

```
TACE_IPL ace(IPL_GRAY);
TByteMatrix Image(128, 128);

//Loading input image

//Writing image to ACE16k
ace << C_LAM1 << Image;

//Using the laplace operator
ace.Laplace(C_LAM2, C_LAM1);

//Reading result image
ace << C_LAM2 >> Image;
```

5.6. L2 Cache

The central component of the Bi-i is a high performance digital signal processor (DSP), which executes Bi-i programs. Bi-i v2 and v301 have a Texas Instruments TMS320C6415 DSP, which has a two-level memory architecture for program and data. The first-level program cache is designated L1P, and the first-level data cache is designated L1D. The Bi-i program cannot access the L1P and L1D caches directly. The program and data memory share the second-level memory, designated L2. A part of the L2 memory can be used as cache, if it is configured so.

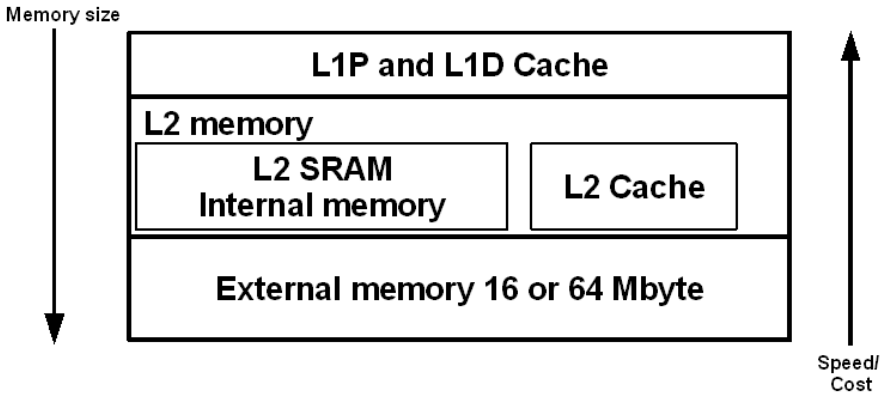

Figure 5.1. Memory Architecture of Bi-i System

Using the L2 cache is useful when the program processes data that is located in the external memory. Without the L2 cache, the speed of the external memory limits the speed of the processing, which can be much slower than when the same operation is performed on the data located in the L2 SRAM. When the L2 cache is enabled, the processing speed for data in the external memory is increased and approaches the speed of processing data in the L2 SRAM.

The TMS320C6415 DSP has 1024 Kbytes of L2 memory. The term L2 cache refers to the part of L2 memory being used as a cache. The L2 cache is a 4-way set associative cache whose capacity is 256 Kbytes with a line size of 128 byte. L2 SRAM is the remaining part of the L2 memory, which can be accessed directly by the Bi-i application.

The Bi-i SDK supports the following L2 modes:

- *L2 SRAM Mode:* The entire L2 memory is mapped as L2 SRAM, thus the external address ranges are not cached. The L2 SRAM is mapped into the address range 0000 0000h to 000F FFFFh . The L2 SRAM mode is the default mode.
- *L2 Cache Mode:* This mode enables the usage of the L2 cache, thus the external address ranges can be cached. The lower 768 Kbyte of L2 memory is mapped as L2 SRAM over the address range 0000 0000h to 000B

FFFFh. The upper 256K byte of L2 memory is mapped as L2 cache over the address range 000C 0000h to 000F FFFFh.

Note

Reading or writing to the L2 address range that is configured as L2 Cache may result in undesired operation of the cache hierarchy. The Bi-i programs must confine L2 accesses to L2 addresses that are mapped as L2 SRAM to ensure correct program operation.

Related documentation from Texas Instruments:

- TMS320C6000 DSP Cache User's Guide (literature number SPRU656).
- TMS320C64x Two-Level Internal Memory Reference Guide (literature number SPRU610).

5.6.1. L2 Mode Selection

When the L2 Cache Mode is selected, 16 MByte ranges of the external memory can be individually enabled for caching. The number of the available address ranges depends on the size of the external memory of the Bi-i system, i.e. one range for 16MB, 4 ranges for 64 MB.

The following continuous parts of the external memory can be enabled for caching:

- *C_P1* - the first part of the SDRAM. Range 8000 0000h - 80FF FFFFh
- *C_P2* - the second part of the SDRAM. Range 8100 0000h - 81FF FFFFh
- *C_P3* - the third part of the SDRAM. Range 8200 0000h - 82FF FFFFh
- *C_P4* - the fourth part of the SDRAM. Range 8300 0000h - 83FF FFFFh

The following steps are needed to enable the L2 cache in the Bi-i application:

- Be sure that the memory range of the L2 cache is not used in the application for any other purpose. The easiest way to do this is to modify the linker command file of the project, for example changing the line "IntRAM: o = 00000800h l = 000FF800h" to "IntRAM: o = 00000800h l = 000BF800h", which disables the general usage of the upper 256 kByte of the L2 memory.
- Include the L2Cache.h file from the /include/SDK/Bii directory.
- Use the SetCacheState function to enable the L2 cache, pass the required external address ranges as parameter.

Example 5.34. L2 Mode Selection

```
#include "SDK/Bii/L2Cache.h"

// This function selects the L2 Cache Mode
// and sets the cacheability of all external address ranges
// C_P1 First part of the SDRAM (16MByte). Range 8000 0000 - 80FF FFFF
// C_P2 Second part of the SDRAM (16MByte). Range 8100 0000 - 81FF FFFF
// C_P3 Third part of the SDRAM (16MByte). Range 8200 0000 - 82FF FFFF
// C_P4 Fourth part of the SDRAM (16MByte). Range 8300 0000 - 83FF FFFF
SetCacheState(C_P1 | C_P2 | C_P3 | C_P4);

// This function selects the L2 SRAM Mode
// The entire L2 memory is mapped as L2 SRAM again
SetCacheState(0);
```

5.6.2. Memory Coherence

The memory system of the DSP supports requests from two sources: CPU and DMA. All peripherals access the memory system by DMA. Generally, if the CPU and a DMA channel share the same cacheable memory region,

the cache and the memory can become incoherent. Consider the system shown in Figure 5.2, "Cache Coherence Problem". Suppose that the CPU accesses a memory location that gets subsequently allocated in cache (1). Later, a peripheral is writing data to this same location that is meant to be read and processed by the CPU (2). However, since this memory location is kept in cache, the memory access hits in cache and the CPU reads the old data from the L2 cache instead of the new data from external memory (3). A similar problem occurs if the CPU writes to a memory location that is cached, and the data is to be read by a peripheral. The data only gets updated in cache, but not in memory, from where the peripheral reads the data. In this case, the cache and the memory are said to be incoherent. Consequently, if a memory location is shared, cached, and has been modified, there is a cache coherence problem.

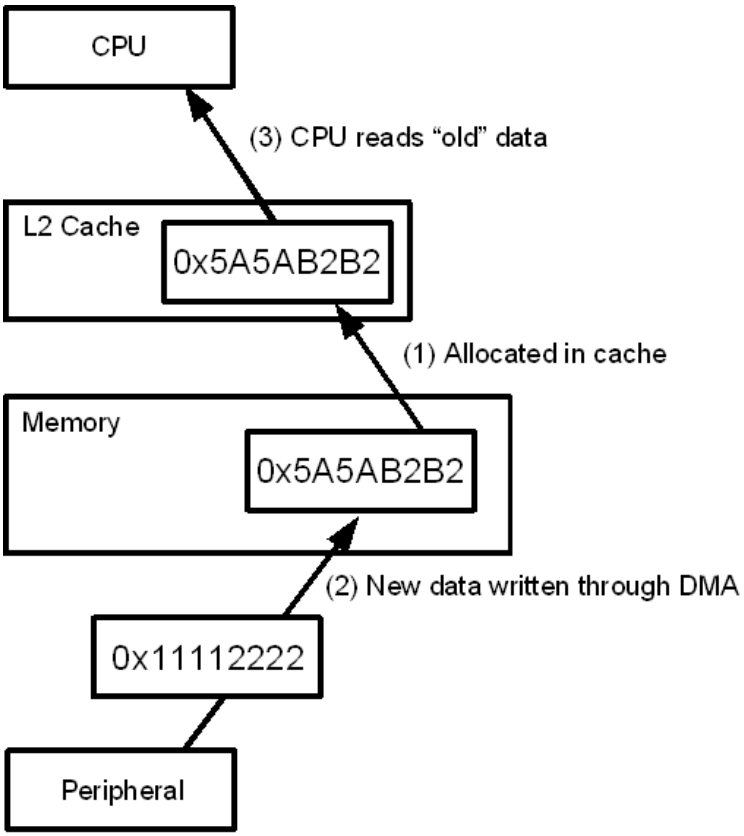

Figure 5.2. Cache Coherence Problem

The memory coherence problem is completely eliminated if the L2 memory is used in SRAM mode, which is the default in the Bi-i SDK. Because of the relatively low speed of the external memory access, this mode is recommended if the application does not process data located in the external memory.

The memory coherence problem does not occur if the L2 cache is enabled, but the peripheral operations use non-cacheable memory locations. This can be either the remaining part of L2 memory (which is not used as cache), or a part of the external memory, which is not enabled for caching. Because the peripheral operations and the CPU use different memory locations, the data must be copied between the different types of operations.

The memory coherence problem can be avoided even if L2 cache is used together with peripheral operations on cacheable memory locations. To do this, special programming steps are required before beginning a peripheral operation. If the affected address range is partially or completely located in the L2 cache, than it must be written back to the original location and invalidated in the cache. The CPU must not use the affected memory range during the peripheral, otherwise it would be read into the L2 cache again.

The WriteBackInvalidate function of the Bi-i SDK writes back the data from the L2 cache to the external memory and invalidates the necessary lines in the L2 cache. Although this function receives word address and word count as input, it always operates on complete cache lines containing 128 bytes and aligned to 128-byte. Thus it is recommended to use the WriteBackInvalidate function for memory ranges aligned to a 128-byte and containing a multiple of 128 bytes.

The memory coherence problem is maintained automatically in the Bi-i SDK for many peripheral operations:

- for synchronous host communication,
- for the LogMsg function both in synchronous and asynchronous host mode,
- for the image read-out from the Ibis sensor, if it is performed by the TIbis::Read function.

However, the user application must handle the memory coherence problem in some scenarios that cannot be maintained automatically in the Bi-i SDK:

- for asynchronous host communication,
- for the image read-out from the Ibis sensor, if it is started by the TIbis::ReadStart function,
- for user defined DMA operations.

See the detailed description of these situations in the next sections.

5.6.2.1. Asynchronous Host Communication

The memory coherence problem can occur because of the host operations of a Bi-i application, if the L2 Cache Mode is enabled and asynchronous host communication is used. (See Section 5.3, “Host Communication”, Section 5.3.1.2, “Asynchronous Communication”.)

The memory ranges affected by a host operation are not limited to the target or the source data of the operation. The Bi-i SDK maintains internal data belonging to the host operations in the .host_request section and also in reserved memory area at the beginning of the external memory. When using asynchronous host operations, all of the affected memory ranges must be considered:

- The lower 16 MByte of the external memory must be non-cacheable, because it contains the reserved area for the host communication.
- The .host_request section must be placed to non-cacheable memory: It either to the L2 SRAM or to a part of the external memory where the L2 cache is disabled.
- The target or source memory ranges must either be non-cacheable, or the WriteBackInvalidate must be called for them before the host operation.

Example 5.35. L2 Cache and Asynchronous Host Communication

```
TByteMatrix Image_1(width,height, NONCACHEABLE_ADDRESS)
TByteMatrix Image_2(width,height, CACHEABLE_ADDRESS)

//The simplest class from IO group
TStd Std(1);

//This function enables the Asynchronous Host Communication
SetAsyncMode(true,TNetTarget());

// This function selects the L2 Cache Mode
// and sets the cacheability of external address ranges
// without the first 16 Mbyte part.
SetCacheState(C_P2 | C_P3 | C_P4);

// This function sends an image
// from a noncacheable memory address
// to the Host computer
TStd << Image_1;

// These functions send an image
// from a cacheable memory address
// to the Host computer
WriteBackInvalidate(Image_2.Data(), Image_2.DataSizeDword());
TStd << Image_2;
```

5.6.2.2. Parallel Reading from Ibis sensor

The TIbis class in the Bi-i SDK represents the Ibis5 sensor of the Bi-i system. See Section 5.4, “TIbis”. The L2 Cache Mode memory coherence problem can occur if the image read-out from the Ibis sensor is parallel with the Bi-i program execution. To avoid this problem, the target of the operation must be non-cacheable, or the affected memory location must be invalidated in the cache with the WriteBackInvalidate function before starting the image read-out.

The code segment below shows a typical scenario for using non-cacheable memory. The current image is copied to a different (cacheable) location for processing, while the new data is read to the original location.

Example 5.36. L2 Cache and Parallel Reading from the Ibis sensor

```
TByteMatrix TemporalTarget(width,height, NONCACHEABLE_ADDRESS)
TByteMatrix Target(width,height, CACHEABLE_ADDRESS)
TIbis Ibis;

// This function selects the L2 Cache Mode
SetCacheState(C_P1 | C_P2 | C_P3 | C_P4);

while(1)
{
Ibis.Shutter();

//This function starts the image reading
Ibis.ReadStart(TemporalTarget);

{
// Parallel code segment processing the Target image
// Keep TemporalTarget intact here
}

//This function waits for the finish of the image reading
Ibis.ReadWait();

//This line transfers the image from the noncacheable memory
//to cacheable memory
Target = TemporalTarget;
}
```

If the target of the image readout is a cacheable memory range, then it must be invalidated in the cache before starting the image read-out.

```
TByteMatrix TemporalTarget(width,height, CACHEABLE_ADDRESS)
...
//This function starts the image reading
WriteBackInvalidate(TemporalTarget.Data(), TemporalTarget.DataSizeDword());
Ibis.ReadStart(TemporalTarget);
...
```

5.6.2.3. Direct Data Transfer

The Direct Transfer of the Bi-i API allows the application running on the host computer to access the memory of the Bi-i system directly. Reading or writing non-cacheable memory ranges eliminates the memory coherence problem, but when accessing a cacheable memory range with Direct Transfer, special care must be taken to avoid it. In the Bi-i program, the affected memory range must be invalidated in the L2 cache with the WriteBackInvalidate function. After this step, the host computer can read or write data. The Bi-i application may access the affected memory range only when the read or write operation of the host computer is finished. This requires a synchronization mechanism between the host and the Bi-i, which is not supported directly in the Bi-i SDK or API.

Example 5.37. L2 Cache and Direct Data Transfer

```
TByteMatrix Image(width,height, CACHEABLE_ADDRESS)

//Processing of the image
for(int j=0; j<Image.Height(); j++)
   for(int i=0; i<Image.Width(); i++)
   {
   Image(i,j) = Image(i,j) % 42;
   }

//This function writes back all parts of the image from the L2 Cache
//and the Host application will read correct data after this function call.
WriteBackInvalidate(Image.Data(), Image.DataSizeDword());
```

5.6.2.4. User Defined DMA

When selecting the L2 Cache Mode, the memory coherence problem can occur if the DMA is started on a cacheable memory range. Follow the steps below to avoid this problem :

- Write back and invalidate the source memory area from the L2 cache with the WriteBackInvalidate function.
- Invalidate the target memory area in the L2 cache with the WriteBackInvalidate function.
- Start the DMA.
- The CPU must not use the source or the target area before finishing the DMA.

5.7. Video Analytics FrontEnd

Video Analytics FrontEnd is a part of the InstantVision Video Analytics (VA) library. An identical functionality is implemented on the Bi-i SDK, as well, for use on the Bi-i v301f model of Bi-i. VA FrontEnd on the SDK runs on the FPGA of the Bi-i v301f, whereas VA FrontEnd in InstantVision runs on the DSP of any models of Bi-i.

This document focuses on the differences between the VA FrontEnd functionality implemented in the Bi-i SDK and the InstantVision libraries. See the InstantVision documentation for a detailed description of VA FrontEnd.

VA FrontEnd in the Bi-i SDK is implemented in the TVAFrontEndF class, whose interface is nearly identical to that of the TVAFrontEnd class of the InstantVision VA library. An important difference is the usage of the Process function. In TVAFrontEndF, the Process function can be used in the same way as in TVAFrontEnd, but it is also possible to split the same functionality into two parts, so that the ProcessStart function can start processing a frame on the FPGA, and the control is returned to the DSP program. The DSP can execute any other tasks, while the VA

FrontEnd processes the frame on the FPGA. The DSP program can call the ProcessWait function to wait for the completion of the VA FrontEnd process on the FPGA. A typical usage of the TVAFrontEndF class is shown in the example below:

Example 5.38. Using the TVAFrontEndF class

```
#include "SDK/Bii/VA/TVAFrontEndF.h"
...
TVAFrontEndF VAFrontEnd(WIDTH, HEIGHT);
TByteMatrix FEinput(WIDTH, HEIGHT);
TBitMatrix FEoutput(WIDTH, HEIGHT);
...
//Set the features of the algorithm
...
while (GoAhead)
  {
  ...
  VAFrontEnd.ProcessStart(FEoutput, FEinput);
  // The DSP can run other tasks here
  VAFrontEnd.ProcessWait();
  // The result in FEoutput can be used beyond this point
  ...
  }
```

The width of the images to be processed is limited to 320 pixels in TVAFrontEndF. Another restriction is that the width must be a multiple of 32.

The TVAFrontEndF class uses DMA transfers on the DSP. This can cause incorrect behavior if the L2 cache is enabled, which is detailed in Section 5.6, “L2 Cache”. Following the rules below will ensure the proper operation of the TVAFrontEndF together with the L2 cache:

- The internal buffers of the object must be located in a non-cacheable address space. The addresses of the internal buffers are arguments of the constructor. The default value for these addresses is zero, which causes dynamic memory allocation.

- The input image passed as an argument to the Process or ProcessStart function must be located in a non-cacheable address space.

5.8. Laser control

The laser control feature is available on the Bi-i v301f model only, when a laser unit is connected to the GPIO port of the camera. The laser unit consist of a laser source and two mirrors that can deflect the laser light horizontally and vertically. The FPGA inside Bi-i v301f controls the laser unit directly, but the DSP program can transfer instructions to the FPGA concerning the laser control:

- The DSP program can switch the laser source on or off.

- The program can pass pairs of coordinates containing the horizontal and the vertical angle. If only one pair is passed, the laser light is directed continuously in the given direction. When more than one pair of coordinates is specified, the laser will flash alternately among all of the different positions at a rate that is not visible to the human eye, so that all of the specified positions seem to be illuminated continuously.

The laser functionality described above is implemented in the TLaserController class. The LaserOn and LaserOff functions switch the laser source on and off, respectively. The SetPosition function can set one or more directions for the laser. This function has several overloaded versions that set the directions in different ways:

- The SetPosition function can accept two numbers specifying the horizontal and the vertical angles for a single direction. It is also possible to pass a matrix with two columns and a number of rows. The number of rows defines the number of laser spots, while the two columns contain the horizontal and the vertical directions.

- Directions can be specified in degrees if float numbers are given. The valid range is -30 to +30 degrees. The values in degrees are converted to a 16-bit digital value internally. It is also possible to specify the 16-bit digital value directly, if unsigned short arguments are passed to the SetPosition function.

The example below shows how to use the TLaserController class.

Example 5.39. Using the TLaserController class

```
#include "SDK/Bii/LaserController/TLaserController.h"
...
TLaserController Laser;

// four laser points initialized to the corners of a square (-10 and +10 degrees)
TFloatMatrix Points(2,4);
Points(0,0) = 10.0f;
Points(1,0) = 10.0f;
Points(0,1) = 10.0f;
Points(1,1) = -10.0f;
Points(0,2) = -10.0f;
Points(1,2) = -10.0f;
Points(0,3) = -10.0f;
Points(1,3) = 10.0f;

// set the laser points
Laser.SetPosition(Points);

// turn the laser light on
Laser.LaserOn();
```

The TLaserController class uses the DMA channels of the DSP. Care must be taken if the L2 cache of the DSP is enabled, because the DMA transfer together with the L2 cache can result in malfunctions. See Section 5.6, “L2 Cache” for more details. When using TLaserController with the L2 cache enabled, the internal buffers of the object must be located in a non-cacheable memory space. The addresses of the internal buffers can be specified as arguments of the constructor. The default value for these arguments is zero, which instructs the constructor to allocate the buffers dynamically.

The LaserUtils.h file contains additional functions for laser control. The most frequently used function is Laser_PixelToAngle, which solves a common conversion task. The position of the objects to be marked with the laser is usually generated in pixel coordinates on the image of the camera. The Laser_PixelToAngle function converts the pixel coordinates to values that can be directly passed to the SetPosition function of the TLaserController class.

5.9. Debug Bi-i application with JTAG

A Bi-i application can be debugged with Code Composer Studio (CCS) through the JTAG interface of the DSP. The next figures shows the location of the connectors in the different Bi-i versions. These figures show the circuit boards inside the Bi-i cameras, which can be seen after removing the back plane.

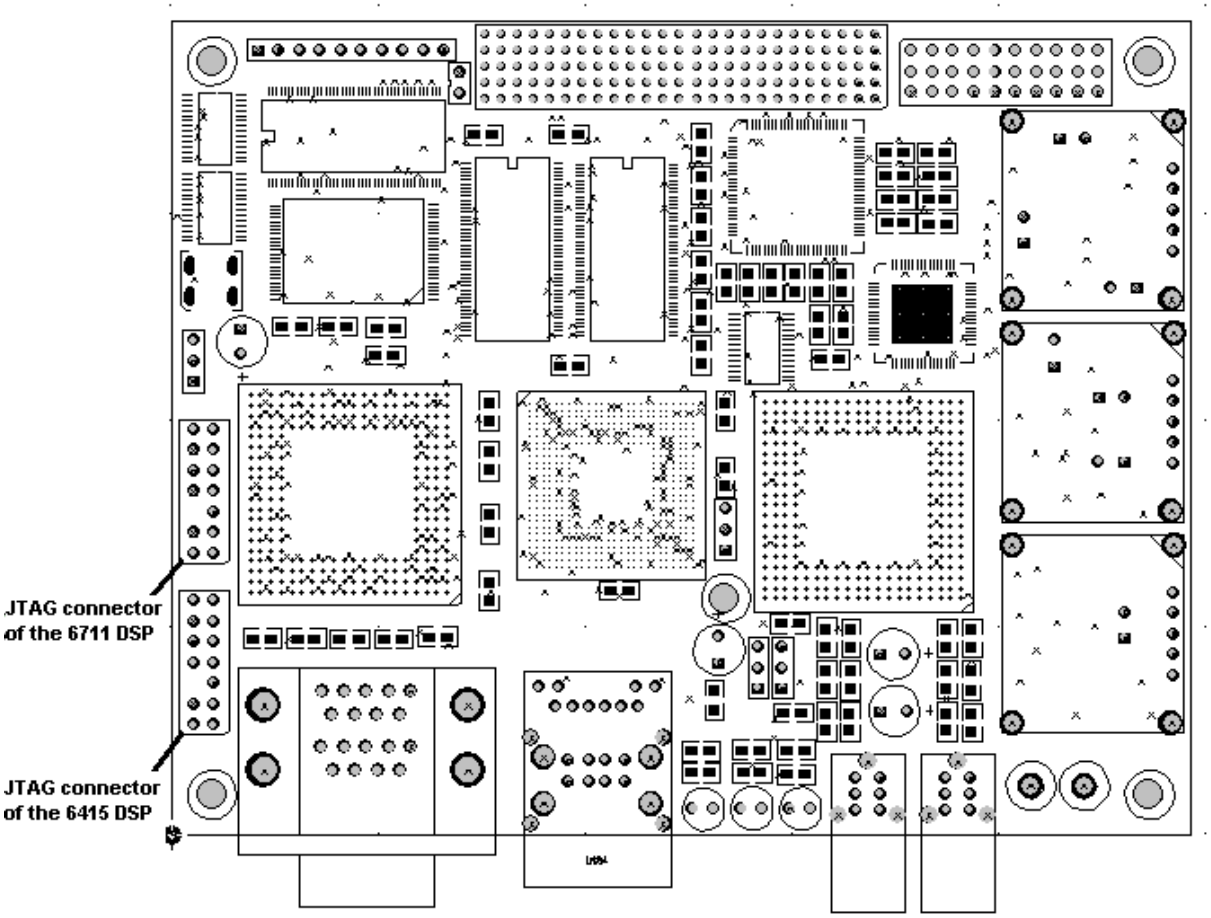

Figure 5.3. JTAG connectors in the Bi-i v2

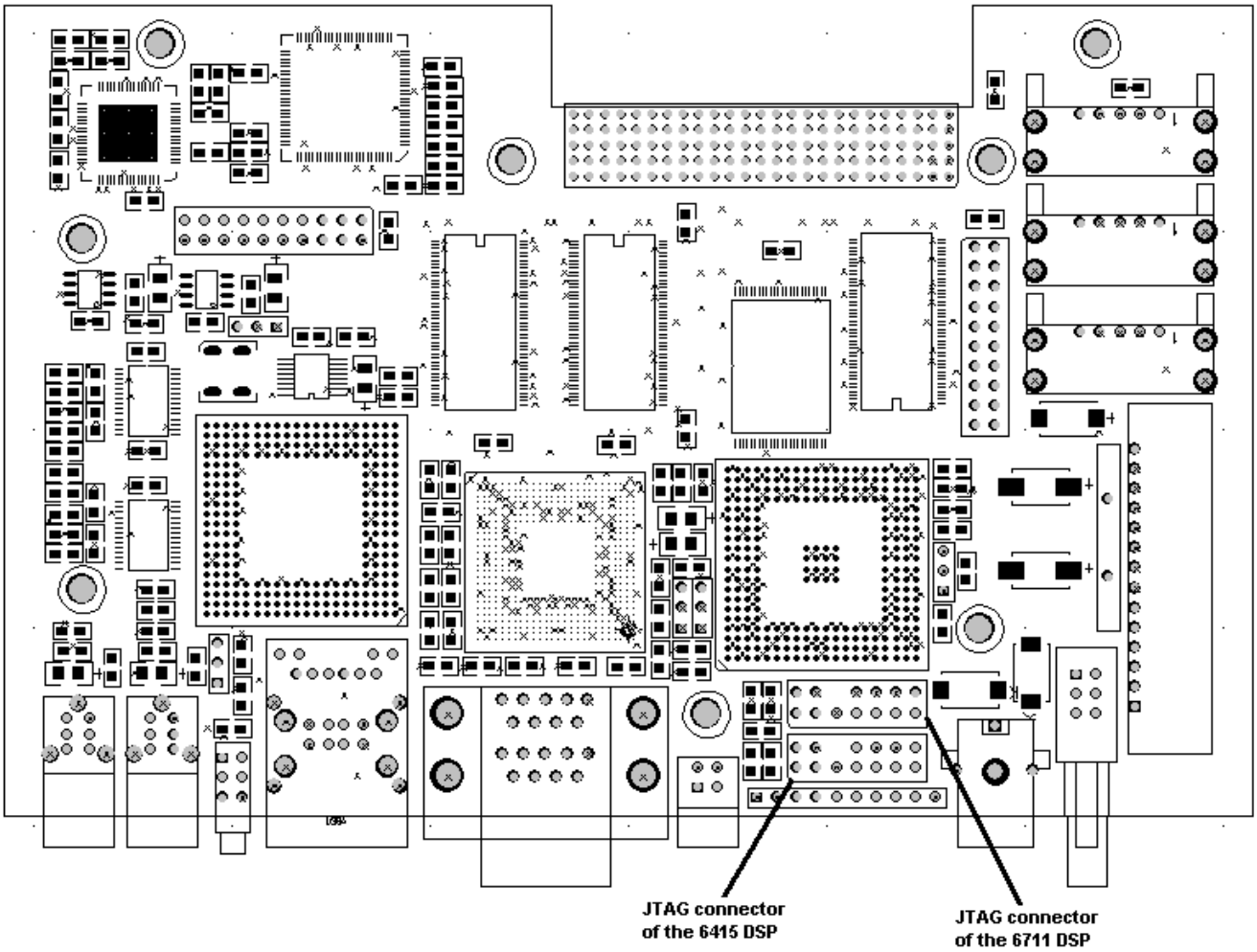

Figure 5.4. JTAG connectors in the Bi-i v301

A compatible JTAG device that supports the TMS320C64XX DSP family is required for debugging. The driver of the JTAG device, which is provided by the manufacturer, must be installed. In addition, the device must be properly set with the CCS Setup tool as follows:

- Add a new board that has a DSP of the C64XX family and a platform matching the type of the JTAG device to your system. A new CPU also appears, which is a part of the board.
- Select the InstantVisionISE\bin\Accessories\Bii6415.gel GEL file as a property of the new CPU.

If more than one device was created in CCS setup, than the CCS Parallel Debug Manager appears each time when CCS is started. The required board must be selected from the open menu to get the usual CCS interface.

To start debugging a Bi-i application, the following steps must be completed:

- In IVConfig, check the "Debug with JTAG" box for the corresponding device.
- In Code Composer studio (CCS), initialize EMIF register with the "GEL / Memory Map and EMIF init / EMIFInit16MB" or "GEL / Memory Map and EMIF init / EMIFInit64MB" menu item, depending on the SDRAM size of the Bi-i system.
- In CCS, download the application to the Bi-i with the File / Load Program menu item.

- In IVRun, select the device (Settings / Run), open the application (File / Open Application) and push the Run button. At this point, the application does NOT start running, which is normal when debugging.
- In CCS, run the application with the Debug/Run menu item.

It is important to follow the order of steps exactly. After completing these steps, the debugging begins with the usual features, such as breakpoints or variable watch. See the CCS documentation for more details.